

MOBPRO - Mobile Programming

Zusammenfassung FS 2019

Maurin D. Thalmann

10. Juni 2019

Inhaltsverzeichnis

1	Android 1 - Grundlagen	4
1.1	Komponenten	4
1.2	Das Android-Manifest	5
1.3	Activities & Aufruf mit Intents	5
1.3.1	Beispielaufruf Expliziter Intent	6
1.3.2	Beispielaufruf Impliziter Intent	7
1.4	Activities & Subactivities	7
1.5	Lebenszyklus & Zustände von Applikationen/Activities	7
1.5.1	Lifecycle einer Applikation	8
1.6	Charakterisierung einer Activity	9
1.6.1	Zustandsänderung - Hook-Methoden	9
1.7	Android - Hinter den Kulissen	10
1.7.1	Android-Security-Konzept	11
2	Android 2 - Benutzerschnittstellen	12
2.1	GUI einer Activity	12
2.2	XML-Layout	13
2.2.1	Constraint-Layout	13
2.2.2	LinearLayout	13
2.3	Ressourcen, Konfigurationen und Internationalisierung	14
2.4	UI-Event-Handling	15
2.4.1	GUI-Events	15
2.4.2	Exkurs: Data Binding	16
2.5	Options-Menü	17
2.6	Adapter-Views	18
2.6.1	AdapterViews & ListActivity	18
2.6.2	android.widget.Spinner	19
2.6.3	android.widget.ListView	20
2.6.4	android.app.ListActivity	20
2.7	ViewModel - Konfigurationswechsel & temporäre Datenspeicherung	21
2.8	Rückmeldungen an den Benutzer	22
2.8.1	Toast	22
2.8.2	Alert-Dialog	22
2.8.3	Notifications (Status-Bar)	25
3	Android 3 - Persistenz & Content Providers	26
3.1	(Shared) Preferences	26
3.1.1	Darstellung User-Preferences	26
3.1.2	PreferenceFragment	28
3.1.3	Default-Präferenzen	28
3.2	Dateisystem	29
3.2.1	Exkurs: Permission-Model	29
3.2.2	Exkurs ff: Runtime Permissions	30
3.2.3	Exkurs ff: Persistenz mit Datei	30
3.3	Datenbank (Room)	31
3.3.1	Room - Code-Beispiele	32
3.3.2	Room - Daten mit Entitäten definieren	33
3.3.3	Room - Beziehungen modellieren	33
3.4	Mit DAOs auf Daten zugreifen	34
3.4.1	Convenience Queries	34
3.4.2	Custom Queries mit @Query	35
3.5	DB-Einträge in einer Liste anzeigen	36
3.6	Content Providers	37
3.6.1	Standard Content Providers	37
3.7	Exkurs : REST-ful Webservices	37

3.8	Content Resolver & Content Provider	38
3.8.1	Zugriff auf Daten über Content Resolver & Query	38
3.9	Eigener Content Provider	39
4	Android 4 - Kommunikation & Nebenläufigkeit	40
4.1	Nebenläufigkeit	40
4.1.1	Android und der Main-Thread	40
4.1.2	Android-Überwachung - ANR (Application Not Responding)	40
4.2	Nebenläufigkeit: AsyncTask	42
4.2.1	Einige Demos aus dem Unterricht	43
4.3	Nebenläufigkeit: Threads	45
4.4	(Backend-) Kommunikation über HTTP	45
4.4.1	HTTP-Requests absetzen	46
4.5	JSON-Webservices mit Retrofit konsumieren	47
4.5.1	JSON Parsing mit GSON	47
4.5.2	Retrofit	47
5	Android 5 - Services & Broadcast Receiver	49
5.1	Service-Komponente	49
5.1.1	Service-Konzept	49
5.1.2	Lebensarten & Lifecycle-Methods eines Service	49
5.1.3	Demo-Code eines Foreground-MusicPlayer-Service	51
5.1.4	Allgemeines Muster & „Stickyness“ von Services	52
5.1.5	Gebundene Services	52
5.1.6	Fortführung Demo Service	54
5.2	Broadcast Receiver	55
5.2.1	Broadcasts versenden	55
5.2.2	Globaler Broadcast Receiver	55
5.2.3	Lokale Broadcasts - „App Message Bus“	56
5.3	Work Manager & Broadcasts	57
6	Android 6 - Intents, Fragments, App-Widgets	58
6.1	Intent Filters	58
6.1.1	Implizite Intents: Daten	58
6.1.2	Implizite Intents: Auflösung von Intents	59
6.2	Fragments	60
6.2.1	Fragments hinzufügen	61
6.3	App-Widgets	61
6.4	App-Design	62
6.5	Usability & Prototyping	62
6.6	Android Jetpack & Support Library (AppCompat)	62
6.7	Publizieren von Android-Apps	63
6.7.1	Signieren von Apps	63
6.7.2	Andere Distributionsmöglichkeiten	63
6.7.3	Test before Release	63
7	Android 7 - Hybrid WebApp	64
7.1	Mobile Web-Anwendungen	64
7.1.1	Charakteristiken für eine Web-App	64
7.1.2	Charakteristiken für eine Hybride App	64
7.1.3	Warum eine App als Web-App entwickeln?	65
7.1.4	Probleme von Webseiten auf Mobil-Geräten	65
7.1.5	Info zu Grundlagen HTML, CSS, JavaScript	65
7.2	Mobile App-Entwicklung mit Ionic	65
7.2.1	Ionic: Projektstruktur	65
7.2.2	Ionic: Seitenaufbau	65
7.2.3	Ionic: How-To's	66
7.3	Hybride Web-Apps mit Cordova (Native Wrapper)	67

7.3.1	Cordova vs. Phonegap	67
7.3.2	Cordova & Ionic	67
8	Android 8 - Entwicklungsansätze	68
8.1	Anforderungen für Entwicklung mobiler Apps	68
8.2	Entwicklung mobiler Apps	68
8.3	Charakteristiken für eine Native App	69
8.4	Charakteristiken für Cross-/JIT-Compile Apps	69
8.5	Native Android-Programmierung	70
8.6	Native iOS-Programmierung	70
8.7	Single vs. Cross Plattform Entwicklung	71
8.8	Szenarien & Fazit	71
8.9	Bester Ansatz?	72
9	Nützliche Links	73
9.1	Android 1 - Grundlagen	73
9.2	Android 2 - Benutzerschnittstellen	73
9.3	Android 3 - Persistenz	73
9.4	Android 4 - Kommunikation & Nebenläufigkeit	74
9.5	Android 5 - Services, Broadcast Receiver	74
9.6	Android 6 - Intents, App-Widgets, Fragments etc.	75
9.7	Web- & Hybrid Mobile Apps mit Ionic	76
9.8	Entwicklungsansätze für mobile Applikationen	76

1 Android 1 - Grundlagen

Informationen zur Androidprogrammierung können stets dem Android Developer Guide entnommen werden unter: *developer.android.com* Apps sollen grundsätzlich gegen das aktuellste API entwickelt werden, aktuell API Level 28 Android 9 „Pie“. Im Gradle-Build-Skript werden deshalb folgende SDK-Versionen festgehalten:

minSdkVersion Mindestanforderung an die SDK, Minimum-Version

targetSdkVersion Ziel-SDK-Version, auf welcher die App lauffähig sein soll

compileSdkVersion Version mit welcher die App (APK) erstellt wird, meist gleich der Target-Version

ART (Android Runtime) verwaltet Applikationen bzw. deren einzelne Komponenten:

- Komponente kann andere Komponente mit Intent-Mechanismus aufrufen
- Komponenten müssen beim System registriert werden (teilweise mit Rechten = Privileges)
- System verwaltet Lebenszyklus von Komponenten: Gestartet, Pausiert, Aktiv, Gestoppt, etc.

1.1 Komponenten

Applikationen sind aus Komponenten aufgebaut, die App verwendet dabei eigene Komponenten (min. eine) oder Komponenten von anderen, existierenden Applikationen.

Name	Beschreibung
Activity	UI-Komponente, entspricht typischerweise einem Bildschirm
Service	Komponente ohne UI, Dienst läuft typischerweise im Hintergrund
Broadcast Receiver	Event-Handler, welche auf App-interne oder systemweite Broadcast-Nachrichten reagieren
Content Provider	Komponente, welche Datenaustausch zwischen versch. Applikationen ermöglicht

Activity entspricht einem Bildschirm, stellt UI-Widgets dar, reagiert auf Benutzer-Eingabe & -Ereignisse. Eine App besteht meist aus mehreren Activities / Bildschirmen, die auf einem „Stack“ liegen.

Basisklasse: *android.app.Activity*

Service läuft typischerweise im Hintergrund für unbeschränkte Zeit, hat keine graphische Benutzerschnittstelle (UI), ein UI für ein Service wird immer von einer Activity dargestellt.

Basisklasse: *android.app.Service*

Broadcast Receiver ist eine Komponente, welche Broadcast-Nachrichten empfängt und darauf reagiert. Viele Broadcasts stammen vom System (Neue Zeitzone, Akku fast leer,...), App kann aber auch interne Broadcasts versenden.

Basisklasse: *android.content.BroadcastReceiver*

Content Provider ist die einzige direkte Möglichkeit zum Datenaustausch zwischen Android-Apps. Bieten Standard-API für Suchen, Löschen, Aktualisieren und Einfügen von Daten.

Basisklasse: *android.content.ContentProvider*

1.2 Das Android-Manifest

AndroidManifest.xml dient dazu, alle Komponenten einer Applikation dem System bekannt zu geben. Es enthält Informationen über Komponenten der Applikation, statische Rechte (Privileges), Liste mit Erlaubnissen (Permissions), ggf. Einschränkungen für Aufrufe (Intent-Filter). Es beschreibt die statischen Eigenschaften einer Applikation, beispielsweise:

(Diese Infos werden bei der App-Installation im System registriert, zusätzliche Infos (Version, ID, etc.) befinden sich im Gradle-Build-Skript (können build-abhängig sein))

- Java-Package-Name
- Benötigte Rechte (Internet, Kontakte, usw.)
- Deklaration der Komponenten
 - Activities, Services, Broadcast Receivers, Content Providers
 - Name (+ Basis-Package = Java Klasse)
 - Anforderungen für Aufruf (Intent) für A, S, BR
 - Format der gelieferten Daten für CP

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="ch.hslu.mobpro.hellohslu">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportRtl="true"
        android:theme="@style/AppTheme">
        <activity
            android:name=".MainActivity"
            android:label="@string/app_name"
            android:theme="@style/AppTheme.NoActionBar">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

Abbildung 1: Beispiel eines Android-Manifests

1.3 Activities & Aufruf mit Intents

Zwischen Komponenten herrscht das Prinzip der losen Kopplung:

- Komponenten rufen andere Komponenten über Intents (= Nachrichten) auf
- Offene Kommunikation: Sender weiss nicht ob Empfänger existiert
- Parameterübergabe als Strings (untypisiert)
- Parameter: von Empfänger geprüft, geparkt & interpretiert (oder ignoriert)
- Keine expliziten Abhängigkeiten → Robuste Systemarchitektur

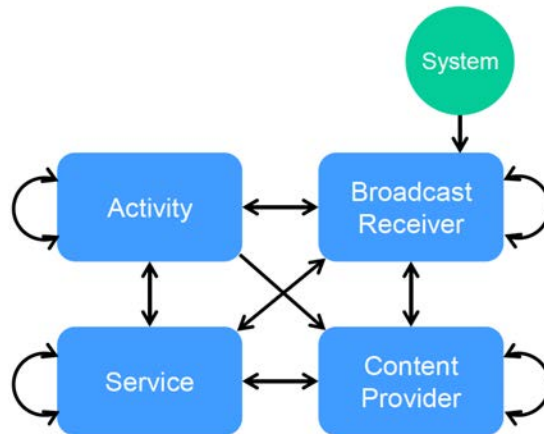


Abbildung 2: Kommunikation zwischen Komponenten mit Intents

Intents werden benutzt, um Komponenten zu benachrichtigen oder um Kontrolle zu übergeben. Es gibt folgende zwei Arten von Intents:

Explizite Intents adressieren eine Komponente direkt

Implizite Intents beschreiben einen geeigneten Empfänger

WICHTIG: Activities müssen immer im Manifest deklariert werden, da sie sonst nicht als „public“ gelten und eine Exception schmeissen. Das geht auch ganz einfach folgendermassen im Manifest unter „application“:

```

1  <activity android:name=".Sender" />
2  <activity android:name=".Receiver" />

```

1.3.1 Beispielaufruf Expliziter Intent

Sender Activity:

```

1  public void onClickSendBtn(final View btn) {
2      Intent intent = new Intent(this, Receiver.class);
3      // Receiver.class ist hier der explizite Empfaenger
4      intent.putExtra("msg", "Hello World!");
5      startActivity(intent);
6  }

```

Receiver Activity:

```

1  public void onCreate(Bundle savedInstanceState) {
2      // ...
3      Intent intent = getIntent();
4      String msg = intent.getExtras().getString("msg");
5      displayMessage(msg);
6  }

```

1.3.2 Beispielaufruf Impliziter Intent

Sender Activity:

```
1 Intent browserCall = new Intent();
2 browserCall.setAction(Intent.ACTION_VIEW);
3 browserCall.setData(Uri.parse("http://www.hslu.ch"));
4 startActivity(browserCall);
```

ACTION_VIEW ist hierbei kein expliziter Empfängertyp, sondern nur eine gewünschte Aktion. Die mitgegebene URL wird auch ein *Call Parameter* genannt. Gesucht ist in diesem Fall eine Komponente, welche eine URL anzeigen/verwenden kann.

1.4 Activities & Subactivities

Activity Back Stack: Activities liegen aufeinander wie ein Stapel Karten, neuste Activity zuoberst und in der Regel ist nur diese sichtbar (Durch Transparenz sind hier Ausnahmen möglich). Durch „back“ oder „finish“ wird die oberste Karte entfernt und man kehrt zur zweitletzten Activity zurück. Mehrere Instanzen derselben Activity wären mehrere solche Karten, das Verhalten kann jedoch konfiguriert werden (z.Bsp. maximal eine Instanz, mehrere Activities öffnen, etc.)

(Sub-)Activities und Rückgabewerte: Eine Activity kann Rückgabewerte einer anderen (Sub-)Activity erhalten.

```
1 // 1. Aufruf der SubActivity mit
2 startActivityForResult(intent, requestId)
3
4 // 2. SubActivity setzt am Ende Resultat mit
5 setResult(resultCode, intent) // intent als Wrapper fuer Rueckgabewerte
6
7 // 3. SubActivity beendet sich mit
8 finish()
9
10 // 4. Nach Beendigung der SubActivity wird folgendes im Aufrufer aufgerufen:
11 onActivityResult(requestId, resultCode, intent)
12 // resultCode: RESULT_OK, RESULT_CANCELLED
```

1.5 Lebenszyklus & Zustände von Applikationen/Activities

Das System kann Applikationen bei knappem Speicher ohne Vorwarnung terminieren (nur Activities im Hintergrund, dies geschieht unbemerkt vom User, die App wird bei Zurücknavigation wiederhergestellt). Eine Applikation kann ihren Lebenszyklus demnach nicht kontrollieren und muss in der Lage sein, ihren Zustand speichern und wieder laden zu können. Applikationen durchlaufen mehrere Zustände in ihrem Lebenszyklus, Zustandsübergänge rufen Callback-Methoden auf (welche von uns überschrieben werden können).

Activity-Zustände:

Zustand	Beschreibung
Running	Die Activity ist im Vordergrund auf dem Bildschirm (zuoberst auf dem Activity-Stack für die aktuelle Aufgabe).
Paused	Die Activity hat den Fokus verloren, ist aber immer noch sichtbar für den Benutzer.
Stopped	Die Activity ist komplett verdeckt von einer andern Activity. Der Zustand der Activity bleibt jedoch erhalten.

1.5.1 Lifecycle einer Applikation

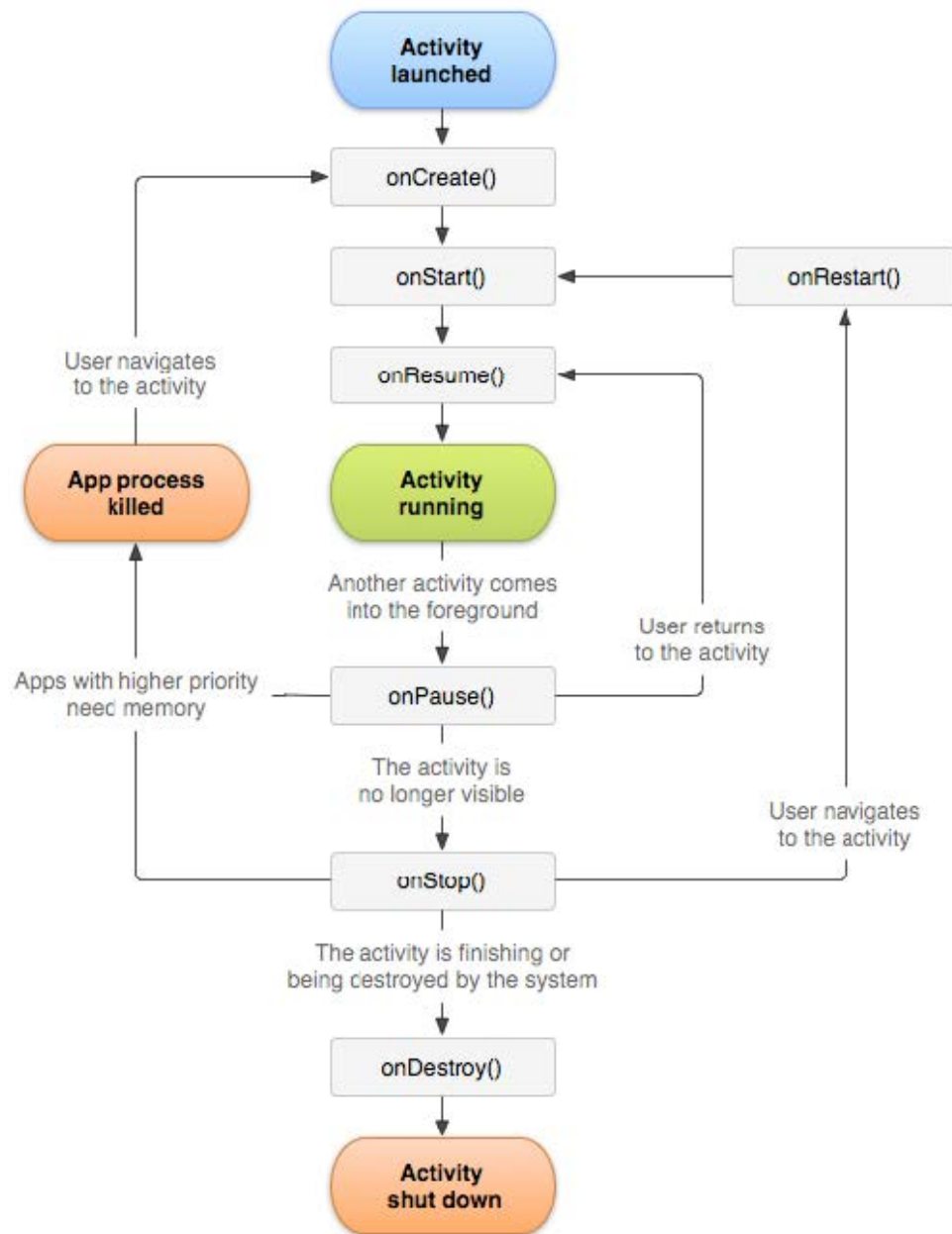


Abbildung 3: Lifecycle einer Applikation

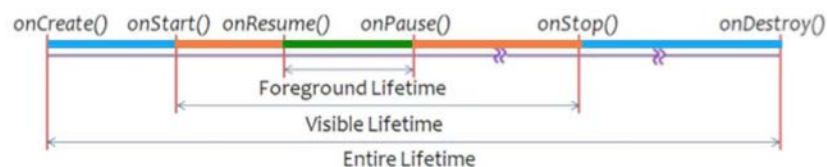


Abbildung 4: Lebenszeiten der einzelnen App-Zustände

1.6 Charakterisierung einer Activity

- Muss im Manifest deklariert werden
- GUI-Controller
 - Repräsentiert eine Applikations-/Bildschirmseite
 - Definiert Seitenlayout und GUI-Komponenten
 - Kann aus Fragmenten (= „Sub-Activities“) aufgebaut sein
 - Reagiert auf Benutzereingaben
 - Beinhaltet Applikationslogik für dargestellte Seite

Beispiel einer Activity:

```
1 public class Demo extends Activity {  
2     // Called when the Activity is first created  
3     public void onCreate(Bundle savedInstanceState) {  
4         super.onCreate(savedInstanceState);  
5         setContentView(R.layout.main); // Definiert Layout und UI  
6     }  
7 }
```

1.6.1 Zustandsänderung - Hook-Methoden

Das System benachrichtigt Activities durch Aufruf einer der folgenden Methoden der Klasse *Activity*:

- void onCreate(Bundle savedInstanceState)
- void onStart() / void onRestart()
- void onResume()
- void onPause() → *bspw. Animation stoppen*
- void onStop()
- void onDestroy() → *bspw. Ressourcen freigeben*

Durch das Überschreiben dieser Methoden können wir uns in den Lebenszyklus einklinken. Immer **super()** aufrufen, sonst wirft es eine Exception.

1.7 Android - Hinter den Kulissen

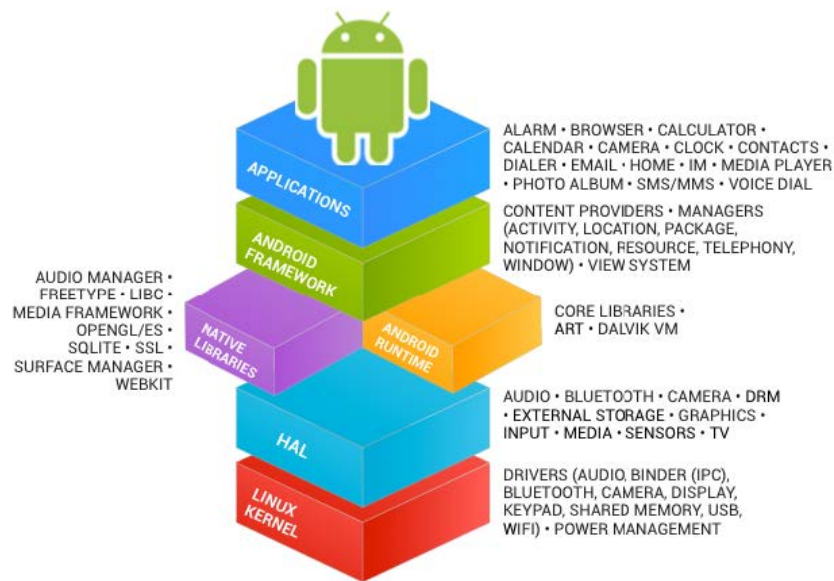


Abbildung 5: Der Android-Stack

- **Linux-Kernel:** OS, FS, Security, Drivers, ...
- **HAL (Hardware Abstraction Layer):** Camera-, Sensor-, ... Abstraktion
- **ART (Android Runtime)**
 - Jede App in eigenem Prozess
 - Optimierte für mehrere JVM auf low-memory Geräten
 - Eigenes Bytecode-Format (Crosscompiling)
 - JIT und AOT Support
- **Native C/C++ Libraries:** Zugriff via Android NDK
- **Android Framework:** Android Java API
- **Applications:** System- und eigene Apps

1.7.1 Android-Security-Konzept

Sandbox-Konzept:

- Jede laufende Android-Anwendung hat seinen eigenen Prozess, Benutzer, ART-Instanz, Heap und Dateisystembereich → jedes App hat eigenen Linux-User
- Das Berechtigungssystem von Linux ist Benutzer-basiert, es betrifft deshalb sowohl den Speicherzugriff wie auch das Dateisystem.
- Anwendungen signieren: erschwert Code-Manipulationen und erlaubt das Teilen einer Sandbox bei gleicher sharedUser-ID
- Berechtigungen werden im Manifest deklariert, kontrollierte Öffnung der Sandbox-Restriktionen

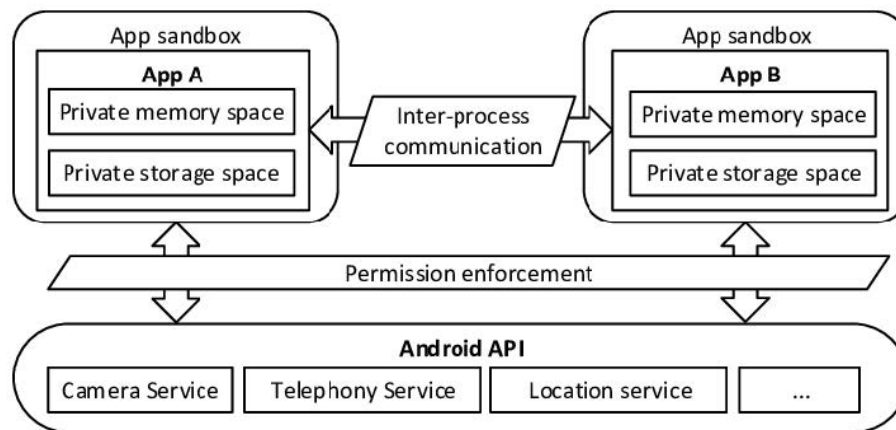


Abbildung 6: Android Security-Modell

2 Android 2 - Benutzerschnittstellen

2.1 GUI einer Activity

GUI wird als XML definiert, der Name resultiert in einer Konstante: **R.layout.xxx**. Diese wird im *onCreate()* einer Activity mit *setContentView()* angegeben.

```
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical"
        android:padding="@dimen/padding">

        <TextView
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:paddingBottom="@dimen/padding"
            android:text="@string/main_title"
            android:textSize="@dimen/textSizeTitle" />

        <TextView
            android:layout_width="match_parent"
```

Abbildung 7: Beispiel eines XML für ein Layout

Je nach Layout müssen die Elemente unterschiedlich konfiguriert werden, was bei der Arbeit mit dem Layout-Editor nicht offensichtlich, aber trotzdem gut zu wissen ist.

Ein Android-UI ist hierarchisch aufgebaut und besteht aus **ViewGroups** (Container für Views oder weitere ViewGroups, angeordnet durch Layout) und **Views** (Widgets). Sollte auf unterschiedlichen Bildschirmgrößen gleich aussehen (Elemente deshalb **relativ** und nicht absolut positionieren)

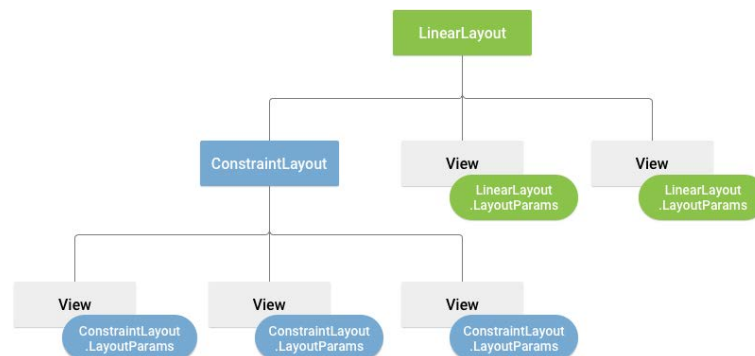


Abbildung 8: Layout-Varianten bei Android

Schachtelung möglich, aber nicht effizient, wenn möglich immer das Constraint-Layout verwenden. Layouts spezifiziert man auf zwei verschiedene Arten:

- **Statisch / Deklarativ (XML)**
- Grundsätzlich in **MOBPRO** verwendet, bietet viele Vorteile (Deklarativ, weniger umständlich als Code, Struktur eminent, Umformungen ohne Rekompilierung möglich...)
 - Deklarative Beschreibung des GUI als Komponentenbaum
 - XML-Datei unter *res/layout*
 - Referenzen auf Bilder/Texte/etc.
 - Typischerweise ein XML pro Activity
- **Dynamisch (in Java)**
- Jedes XML hat eine korrespondierende Java-Klasse, XML → Java = *Inflating*
 - Aufbau und Definition des GUI im Java-Code
 - Normalerweise nicht nötig: die meisten GUIs haben fixe Struktur
 - Änderung von Eigenschaften während Laufzeit ist normal (Bsp. Visibility, Ausblenden einer View, wenn nicht benötigt)

2.2 XML-Layout

- Jedes Layout ist ein eigenes XML-File
 - Root-Element = View oder ViewGroup
 - Kann Standard- oder eigene View-Klassen enthalten
- XML können mit Inflater „aufgeblasen“ bzw. instanziiert werden, damit eigene wiederverwendbare Komponenten/Templates/Prototypen erzeugt werden können
- Innere Elemente können unterhalb eines Parents via View-ID referenziert werden (*findViewById()*)
- Debugging mit dem Layout-Inspector

2.2.1 Constraint-Layout

- Erstellung von komplexen Layouts, ohne zu schachteln
- Elemente werden relativ mit Bedingungen platziert
 - zu anderen Elementen
 - zum Parent-Container
 - Element-Chains (spread/pack)
- Layout-Hilfen (Hilfslinien, Barriers)

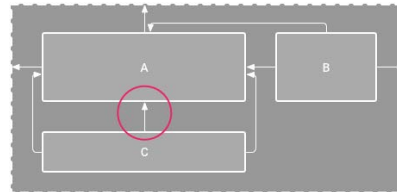


Abbildung 9: Constraint Layout

2.2.2 LinearLayout

- Reiht Elemente neben-/untereinander auf
 - kann geschachtelt werden, um Zeilen/-Spalten zu formen (nicht zu tief, sonst schlechte Performanz)
- Eigenschaften: (*orientation, gravity, weighthSum, etc.*)
- Layout-Parameter für Children
 - layout_width, layout_height
 - layout_margin...
 - layout_weight, layout_gravity

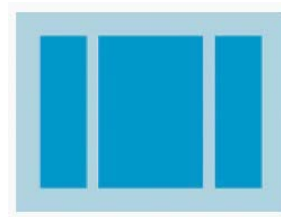


Abbildung 10: LinearLayout

Warum nutzt man trotzdem noch LinearLayout?

- Nach wie vor einfachste Lösung für Button- oder Action-Bars („flow semantik“) und einfache Screens
- Kaum Konfiguration nötig, robust
- Für scrollbare Listen mit dynamischer Anzahl Elemente besser *ListView* verwenden (siehe Adapter-Views)
- Einsatz mit Bedacht durchaus sinnvoll

Es gibt noch die **ScrollView**, deren Nutzung vertikales Scrollen bei zu grossen Layouts erlaubt, sie kann jedoch nur ein Kind haben und enthält typischerweise das Top-Level-Layout einer Bildschirmseite.

Pixalangaben (*Typischerweise werden Angaben in dp verwendet, ausser sp bei Schriftgrössen.*)

- **dp - density-independent:**
Passen sich der physischen Dichte des Screens an, dp passen sich gegenüber den realen Dimensionen eines Screens und dessen Verhältnisse an.
- **sp - scale-independent:**
Ähnlich der dp-Einheit, passt sich jedoch der Schriftskalierung des Nutzers an.
- **px - Pixels:**
Passen sich der Anzahl Pixel eines Bildschirms an, deren Nutzung wird nicht empfohlen.

2.3 Ressourcen, Konfigurationen und Internationalisierung

Ressourcen sind alle Nicht-Java-Teile einer Applikation und sind im `/res`-Verzeichnis abgelegt, sogenannte ausgelagerte Konstanten-Definitionen. Sie werden im Layout und Java-Code über die **automatisch generierte R-Klasse** mit ID-Konstanten (int) referenziert. Kontextabhängige Ressourcen sind möglich z.Bsp. für Sprache, Gerätetyp, Orientierung, ...

Beispiele: Strings, Styles, Colors, Dimensionen, Bilder (drawables), Layouts (portrait, landscape), Array-Werte (z.Bsp. für Spinner) und Menü-Items

```
<TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginBottom="@dimen/marginBottom"
    android:background="@color/sectionBackground"
    android:padding="@dimen/padding"
    android:text="@string/main_section1"
    android:textColor="@color/sectionText" />
```

(a) Referenz in XML mit @

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
}
```

(b) Referenz in Code über R-Klasse, diese wird beim Build automatisch generiert

Für verschiedene Systemkonfigurationen benötigt es unterschiedliche Ausprägungen einer Ressource, beispielsweise:

- **Internationalisierung:** komplette/teilweise Übersetzung, für diese werden unterschiedliche Ordner je nach Land/Sprache und separate .xml angelegt
- **Auflösungsklassen:** `ldpi` (120dpi), `mdpi` (160dpi), `hdpi` (240dpi), `xhdpi` (320dpi)
- **Orientierung** des Displays: `landscape` / `portrait`
- Verschiedene **HW-Modelle:** HTC, Samsung, Sony, LG, ...

Default-Verzeichnisse sind innerhalb von `res/` angelegt: `drawable`, `layout`, `menu`, `values`, ...

Bei spezifischen Konfigurationen werden meist Kopien der Default-Verzeichnisse/Ordner mit einem Suffix angelegt, bspw. `res/strings-de-rCH`, in welchen dann die Ressourcen (XML) erneut angelegt werden.

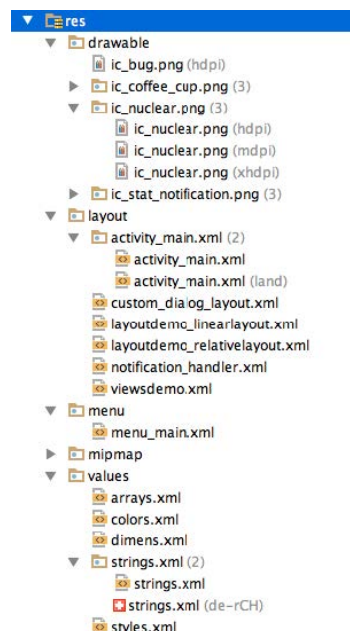


Abbildung 12: Beispiel der Default-Ressourcen

2.4 UI-Event-Handling

- Jedes View-Element hat eine entsprechende Java-Klasse (auch View-Groups!)
→ Layout könnte auch dynamisch in Java programmiert werden
- APIs der einzelnen View-Klassen sind hier oder unter „Nützliche Links“ genauer beschrieben

```
<TextView
    android:id="@+id/message_label"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />

// Show message on dedicated text view
private void displayMessage(String message) {
    TextView label = (TextView)findViewById(R.id.message_label);
    label.setText(message);
}
```

Abbildung 13: ID im Layout erfassen und Referenz im Code

2.4.1 GUI-Events

- **Observer/Listener:** einen Listener für ein entsprechendes Event bei der View registrieren, bspw. bei Button myButton:
myButton.setOnClickListener(listener)
- verschiedenste Event- und Listener-Typen:
OnClickListener, OnLongClickListener, OnKeyListener, OnTouchListener, OnDragListener, ...
→ public static Interfaces der Klasse View

Ziel: Auf Klick-Event eines Buttons reagieren

- Button muss eine ID haben im layout.xml
- Registrierung eines Listeners an die View (Button) im Code:

```
1 Button button = (Button) findViewById(R.id.question_button_done);
2 button.setOnClickListener(new OnClickListener() {
3     @Override
4     public void onClick(View v) {
5         // handler code
6         buttonClicked();
7     }
8 });
```

onClick-Event-Registrierung in XML

```
<Button
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:onClick="increaseInternalCounter"
    android:layout_marginBottom="@dimen/marginBottom"
    android:text="@string/main_increaseInternalCounter" />
```

Abbildung 14: Definition onClick-Handler im Layout → so nur für onClick-Events

```
1 // Implementierung onClick-Handler-Methode in der Activity
2 public void increaseInternalCounter(View button) {
3     // ... handler code ...
4 }
```

2.4.2 Exkurs: Data Binding

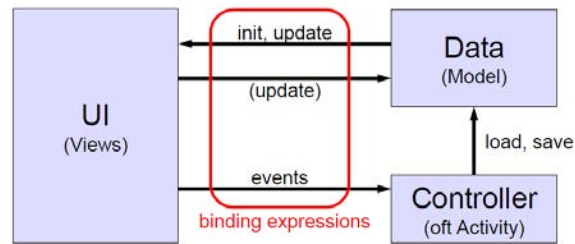


Abbildung 15: Modell für Data Binding

Data Binding: separiert UI und Daten, synchronisiert UI mit Daten (1-, resp. 2-way-binding), verwendet «binding expressions» mit @.. Syntax im Layout-File, um View-Attribute zu initialisieren. Anbei ein Beispiel (auskommentiert):

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <layout xmlns:android="http://schemas.android.com/apk/res/android">
3   <data>
4     <variable name="model" type="org.example.MyModel"/>
5   </data> // Definition der Layout-Variablen
6   <LinearLayout ...>
7     <Button
8       android:id="@+id/button"
9       ...
10      android:enabled="@{model.user.role == 'admin'}"
11      android:text="@{model.buttonText}" // Data Binding (1-way)
12      ...
13      android:onClick="@{() -> model.increaseClickCount()}" /> // Event Binding
14    <EditText
15      android:id="@+id/input"
16      ...
17      android:text="@={model.inputText}" /> // Data Binding (2-way)
18    </LinearLayout>
19  </layout>
20
21  protected void onCreate(Bundle savedInstanceState) {
22    super.onCreate(savedInstanceState);
23    ActivityMainBinding binding = DataBindingUtil setContentView(...);
24    model = new MainModel();
25    model.load();
26    binding.setModel(model);
27    // Binden der Layout-Daten auf effektive Daten
28    // z.B. ViewModel mit Observables
29  }
```

2.5 Options-Menü

- Android-Apps können oben rechts ein Menü mit Optionen anbieten
- Erzeugung durch Aufruf *Hook* in der Activity-Klasse:

`onCreateOptionsMenu(Menu menu)`

- Hier kann ein Menü mit Einträgen bestückt werden
- `MenuInflater` + XML benutzen oder Java oder beides

- Beim Klick auf Eintrag Aufruf eines anderen Hooks:

`onOptionsItemSelected(MenuItem item)`

Für ein Options-Menü muss eine .xml-Datei (Bsp. `main_menu.xml`) im Ordner `res/menu` angelegt werden. Danach werden Informationen folgendermassen eingetragen:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:tools="http://schemas.android.com/tools" tools:context=".MainActivity">
  <item
    android:id="@+id/main_menu_finish"
    android:title="@string/menu_finish">
  </item>
  <item
    android:id="@+id/main_menu_startAllViews"
    android:title="@string/menu_startViewsDemo">
  </item>
</menu>
```

(a) Menü und Items in XML definieren

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    super.onCreateOptionsMenu(menu);
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.menu_main, menu);
    return true;
}
```

(b) Menü mit `MenuInflater` aufblasen

Um bspw. einen String in einem Menüpunkt einzufügen, gibt es drei verschiedene Möglichkeiten:

```
menu.add(Menu.NONE, 239, Menu.NONE, "Menu Item 1");
menu.add(Menu.NONE, 333, Menu.NONE, getString(R.string.menu_mail));
menu.add(Menu.NONE, 923, Menu.NONE, R.string.menu_server);
```

Abbildung 17: Möglichkeiten zum Einlesen eines Strings

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    if (super.onOptionsItemSelected(item)) {
        return true; // handled by super implementation
    }
    switch (item.getItemId()) {
        case R.id.main_menu_finish:
```

Abbildung 18: Event-Handling: Selektierung

2.6 Adapter-Views

Behandelt wird hier nur das synchrone Laden von kleinen/schnellen Datenquellen, für asynchrones Laden von langsamen/grossen Datenquellen konsultiere Doku über **Loaders**.



Abbildung 19: Aufgabe des Adapters

- Adapter → Verbindung zwischen Datenquelle und GUI
- Zapft *Datenquelle* an und beliefert *AdapterView*
- Erzeugt (Sub-)Views pro gefundenes Datenelement
- Transformiert Daten ggf. in benötigtes Zielformat
- Datenquellen:
String-Array, String-Liste, Bilder, Datenbank, ...

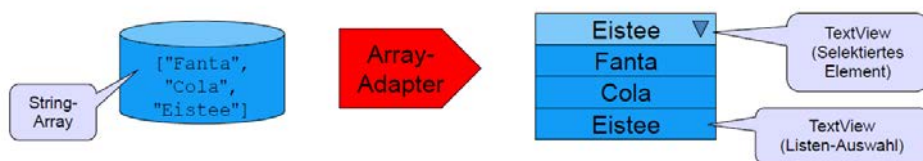


Abbildung 20: Beispiel eines ArrayAdapter

- Bindet irgend ein Array oder Liste mit beliebig getypeten Elementen an irgend eine AdapterView
- Für jedes Daten-Element wird eine SubView erzeugt
- **Default:** Erstellt `TextView` mit `element.toString()`-Wert

```
String[] myArray = new String[]{"Fanta", "Cola", "Eistee"};
ArrayAdapter<String> adapter =
    new ArrayAdapter<String>(this, android.R.layout.simple_list_item_checked, myArray);
this.setListAdapter(adapter);
```

Abbildung 21: Beispiel einer AdapterView

2.6.1 AdapterViews & ListActivity

- **AdapterViews:** spezielle View-Klassen
 - Sind für Zusammenarbeit mit Adaptern optimiert (Bsp. `ListView`, `GridView`, `Gallery`, `Spinner`, `Stack`, ...)
 - Füllen Teile von sich mit von Adaptern erzeugten Views
 - Leiten ab von `android.widget.AdapterView<T> extends android.widget.Adapter<T>`
- Spezielle Activity: **ListActivity**
 - Vordefiniertes Layout (enthält eine `ListView`, kein XML nötig)
 - Vordefinierte Callbacks (bei Auswahl einer List-Entry)
 - Bietet Zugriff auf aktuelle Selektion / Datenposition

2.6.2 android.widget.Spinner

- ComboBox oder DropDown-List genannt (weitere Alternative: AutoCompleteTextView)
- Zeigt ein ausgewähltes Element, bei Klick erscheint ein Auswahlmenü
- 2 Varianten, um Daten auf Spinner zu setzen:
 - Im Code mit Adapter:
`spinner.setAdapter(myAdapter)`
 - Im XML mit Angabe einer String-Array-ID:
`android:entries="@array/spinnerValues"`
- Listener setzen für Behandlung der Auswahl:
`spinner.setOnItemSelectedListener(...)`

Demo: Spinner (Siehe Übung 2)

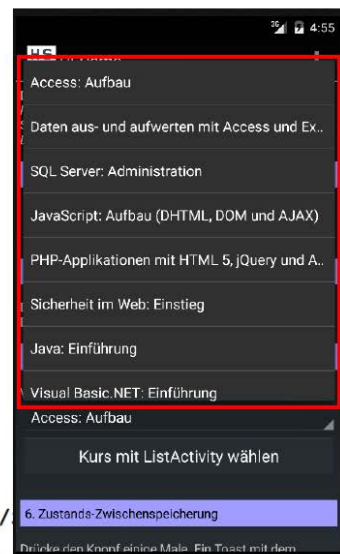
■ layout.xml

```
<Spinner
    android:id="@+id/main_spinner"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:entries="@array/itCourses"
    android:prompt="@string/main_spinnerPrompt" />
```

Beachte: Daten werden aus XML-Ressource geholt

■ arrays.xml

```
<resources>
    <string-array name="itCourses">
        <item>Access: Aufbau</item>
        <item>Daten aus- und aufwerten mit Access und Excel</item>
        <item>SQL Server: Administration</item>
        <item>JavaScript: Aufbau (DHTML, DOM und AJAX)</item>
        <item>PHP-Applikationen mit HTML 5, jQuery und AJAX</item>
        <item>Sicherheit im Web: Einstieg</item>
        <item>Java: Einführung</item>
    
```



■ In der Activity-Klasse

```
spinner.setOnItemSelectedListener(new OnItemSelectedListener() {
    public void onItemSelected(AdapterView<?> parent, View view, int position, long id) {
        String selectedItem = (String) parent.getItemAtPosition(position);
```

Position der View in „ParentView“

Zeilen-ID des gewählten Werts bei DB-Query

Abbildung 22: Übungs-Demo aus der Vorlesung SW02 - Spinner

2.6.3 android.widget.ListView

- Liste von Views/Items, die zur Auswahl stehen
- Braucht viel Platz! Meist wird ihr der ganze Bildschirm zugeteilt
- i.d.R. zusammen mit `ListActivity` verwendet, Verwendung:
 1. Navigiere zu eigener `ListActivity`
 2. Auswahl → Resultat setzen → finish
 3. Auswertung des Rückgabewert im Caller
- Konzeptionell identisch zum Spinner, jedoch andere Darstellung auf UI
 - Verwendungsentscheid:
 - * Kurze Liste → Spinner
 - * (Sehr) lange Listen → `ListView` / `ListActivity`
 - * Kennt der User die möglichen Auswahlwerte → `AutoCompleteTextView`
 - Adapter- / Datendefinition grundsätzlich bei beiden gleich (d.h. im Code oder durch XML-Array)
 - Auswahlmodus: `setChoiceMode(ListView.CHOICE_MODE_*)`
→ Single- / Multiselection

2.6.4 android.app.ListActivity

- Spezielle Activity zur Darstellung einer `ListView`
- Vordefiniertes Layout (full-screen Liste)
 - `setContentView(...)` muss nicht aufgerufen werden
 - Aufruf i.d.R. mit `startActivityForResult(...)`
 - Vordefinierte vererbte Konfigurationsmethoden
 - * `setListAdapter(adapter)` setzt Daten für die Liste
 - * `getListView()` erlaubt Zugriff auf `ListView`-Instanz (anstelle von `findViewById(..)` + Casten)
- Callback bei der Auswahl
 - `onListItemClick(parentView, view, position, id)`
Wird bei Auswahl aufgerufen (muss in Subklasse überschrieben werden, keine Listener-Registrierung nötig)

Demo: ListView & ListActivity (Siehe Übung 2)

- Activity-Klasse (erbt von `ListActivity`!)
- Initialisierung der `ListActivity` mit Daten

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    // Attention: We do NOT set a layout! - ListActivity has already defined a layout.
    String[] courses = getResources().getStringArray(R.array.itCourses);
    ArrayAdapter<String> adapter =
        new ArrayAdapter<String>(this, android.R.layout.simple_list_item_checked, courses);
    this.setAdapter(adapter);
    ListView listView = getListView();
    listView.setChoiceMode(ListView.CHOICE_MODE_SINGLE);
}
```



- Reagieren auf Selektion

```
@Override
protected void onListItemClick(ListView parent, View view, int position, long id) {
    // define return value
    Intent result = new Intent();
    String selectedItem = (String) parent.getItemAtPosition(position);
    result.putExtra(EXTRA_CLASS_KEY, selectedItem);
    // set return value
    setResult(RESULT_OK, result);
    // finish the activity
    finish();
}
```

Position der View in „ParentView“

Zeilen-ID des gewählten Werts bei DQ-Query

Abbildung 23: Übungs-Demo aus der Vorlesung SW02 - `ListView` / `ListActivity`

2.7 ViewModel - Konfigurationswechsel & temporäre Datenspeicherung

Bei jedem Konfigurationswechsel (z.B. Änderung Bildschirmorientierung) wird die aktuelle Activity-Instanz zerstört und neu aufgebaut. Dabei besteht das Problem des **Zustandsverlusts**. Der Zustand aller Views mit einer ID (mit einigen Ausnahmen) wird automatisch gesichert und wiederhergestellt. Der **inhärente Zustand**, alles was nicht sichtbar und in Feldern gespeichert ist, geht jedoch verloren. Um entgegenzuwirken, kann ein **ViewModel** verwendet werden.

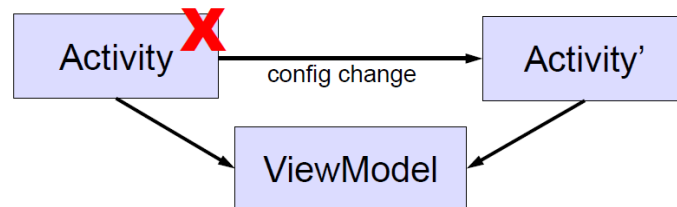


Abbildung 24: Position des ViewModels in der temp. Datenspeicherung

- Kapselt UI-Daten so, dass diese bei einer Konfigurationsänderung einer Activity in-memory erhalten bleiben (Für den Fall eines App-Kills müssen Daten immer noch persistiert werden)
- Lebensdauer mit der Activity gekoppelt
- Weniger Aufwand für Behandlung von Konfigurationsänderungen

```

dependencies {
    ...
    implementation 'androidx.lifecycle:lifecycle-extensions:2.0.0' // ViewModel and LiveData
    ...
}

public class MainViewModel extends ViewModel {
    private int counter = 0;

    public int incrementCounter() { return ++counter; }

    public int getCounter() { return counter; }
}

// in MainActivity
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    viewModel = ViewModelProviders.of(this).get(MainViewModel.class);
    counterLabel = findViewById(R.id.main_label_counter);
    updateCounterLabel();
}

// called on button click (see main.xml)
public void increaseInternalCounter(View button) {
    viewModel.incrementCounter();
    updateCounterLabel();
}
  
```

Zusätzliche Gradle dependency für ViewModel und Lifecycle Management

ViewModel = normales POJO, ggf. mit Handler-Methoden

Wäre noch viel einfacher mit DataBinding! (out-of-scope)

Erzeuge oder hole ViewModel-Instanz für diese Activity-Lebenszyklus-Instanz

Initialisierung UI aus ViewModel

Demo

Abbildung 25: Übungs-Demo aus der Vorlesung SW02 - ViewModel

2.8 Rückmeldungen an den Benutzer

2.8.1 Toast

- Kurze Rückmeldung (Popup) an den Benutzer, keine Interaktion möglich, verschwindet nach gewisser Zeit.
- Konfiguration: Text, Layout, Anzeigzeit (kurz/lang), Ort (gravity)
- Toasts mit eigenem Layout werden mit CustomToastView erstellt

Beispielcode zur Erstellung von Toasts:

```
1 // Default-Toast: Einzeiler
2 Toast.makeText(getApplicationContext(), "Das ist..", Toast.LENGTH_LONG).show()
3                                     // LENGTH: Nur LONG oder SHORT
4                                     // Kontext: meistens "this"
5
6 // Toast mit anderem Anzeigort:
7 Context context = getApplicationContext();
8 Toast toast = Toast.makeText(context, "Toast links oben!", Toast.LENGTH_LONG);
9 toast.setGravity(Gravity.TOP|Gravity.START, 0, 0); // (x,y) Offset
10 toast.show()
```

2.8.2 Alert-Dialog

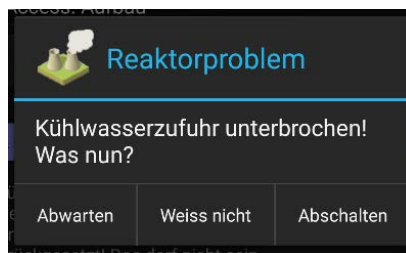


Abbildung 26: Beispiel eines Alert-Dialogs

- Fenster mit Interaktionsmöglichkeiten für den Benutzer
 - Information / Eingabe von Daten
 - Interaktion möglich
 - Buttons: positive, neutral, negative
- Vorteile
 - Kaum Einschränkungen in puncto Darstellung
 - Vorbereitet für die Anzeige von Daten
 - Verschwindet erst, wenn sie vom Benutzer quittiert wurde
- Konfiguration: Buttons, Titel, Icon, Nachricht
Inhalt: Liste von Items oder eigene View

- Vorgehen beim Erstellen eines Alert-Dialog mit Builder-Muster
 1. Builder erstellen: `new AlertDialog.Builder(this)`
 2. Builder konfigurieren:
`setXXX + Registrierung von ClickListeners`
 3. Dialog erstellen: `Dialog dialog = builder.create()`
 4. Dialog anzeigen: `dialog.show()`
- Anzeige von Dialogen ist **immer asynchron!**
Bei `show()` wird nicht gewartet, kein Rückgabewert
→ Behandlung von Benutzerselektion mit Listener

```
AlertDialog.Builder dialogBuilder = new AlertDialog.Builder(this);
dialogBuilder.setTitle("Reaktorproblem")
    .setIcon(R.drawable.ic_nuclear)
    .setMessage("Kühlwasserzufuhr unterbrochen!\nWas nun?")
    .setPositiveButton("Abschalten", new OnClickListener() {
        public void onClick(DialogInterface dialog, int which) {
            Toast.makeText(getApplicationContext(),
                "Reaktor wird abgeschaltet...",
                Toast.LENGTH_LONG).show();
        }
    }).setNeutralButton("Weiss nicht", new OnClickListener() {
        public void onClick(DialogInterface dialog, int which) {
            Toast.makeText(getApplicationContext(),
                "Problem an Support weitergeleitet...",
                Toast.LENGTH_SHORT).show();
        }
    }).setNegativeButton("Abwarten", new OnClickListener() {
        public void onClick(DialogInterface dialog, int which) {
            // do nothing
        }
    });
return dialogBuilder.create();
```

Abbildung 27: Beispiel eines AlertDialog aus Vorlesung

Alert-Dialog mit Auswahl-Daten

- Titel, Icon, usw. wie gehabt
- Neu: Daten (Array) setzten
 - Methode `setItems(...)`
 - Inkl. ClickListener
 - Toast mit Wahl anzeigen!



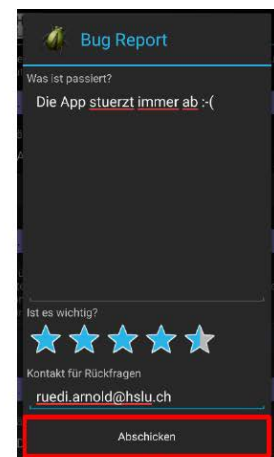
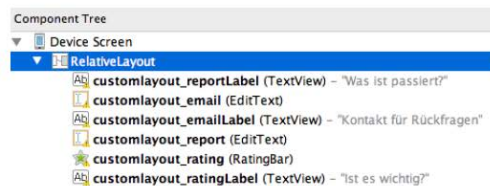
```
final String[] items = {"Kaffee", "Tee", "Wasser"};
AlertDialog.Builder dialogBuilder = new AlertDialog.Builder(this);
dialogBuilder.setTitle("Wähle ein Getränk")
    .setIcon(R.mipmap.ic_launcher)
    .setItems(items, new OnClickListener() {
        public void onClick(DialogInterface dialog, int itemPos) {
```

Handler für
Selection

Abbildung 28: Beispiel mit Auswahl-Daten

Alert-Dialog mit eigenem Layout

- Layout.xml „aufblasen“ & setzen



```
private Dialog createCustomLayoutDialog() {
    final View customView = LayoutInflater.from(this).inflate(
        R.layout.custom_dialog_layout, null);
    AlertDialog.Builder dialogBuilder = new AlertDialog.Builder(this);
    dialogBuilder.setTitle("Bug Report").setIcon(R.drawable.ic_bug)
        .setView(customView)
        .setPositiveButton("Abschicken", new OnClickListener() {
            public void onClick(DialogInterface dialog, int which) {
```

Abbildung 29: Beispiel mit eigenem Layout

Ein (offener) Dialog gehört zum Zustand einer Activity, ist ein Dialog noch geöffnet bei einem Konfigurationswechsel, dann wird dieser nicht gespeichert und auch nicht wiederhergestellt! Deshalb sollten Dialoge als `DialogFragment` implementiert werden. Der Zustand des Dialogs wird dann vom `FragmentManager` korrekt mit Lifecycle und Activity synchronisiert (save/restore)

Für den Moment: Ein **Fragment** ist ein wiederverwendbarer „UI Schnippsel“ mit eigenem Zustand und Lifecycle.

2.8.3 Notifications (Status-Bar)

- Persistente Nachricht
 - Kurze Ticker-Nachricht in der Status-Bar
 - Danach persistente Anzeige im Notification Window
 - Bei Auswahl erfolgt Aufruf einer definierten Activity
- Vorteile:
 - Nachricht bleibt erhalten bis vom Nutzer quittiert
 - Beliebig komplexe Behandlung, da Start einer Activity
- Nachteil:
 - Etwas komplexere Mechanik wegen `PendingIntent`

Demo: Notification

■ Code verwendet u.a.:

- `AlertDialog.Builder`
- `Notification.Builder`
- `PendingIntent`
- `NotificationManager`
- Eigene dedizierte Activity für Darstellung der Nachricht

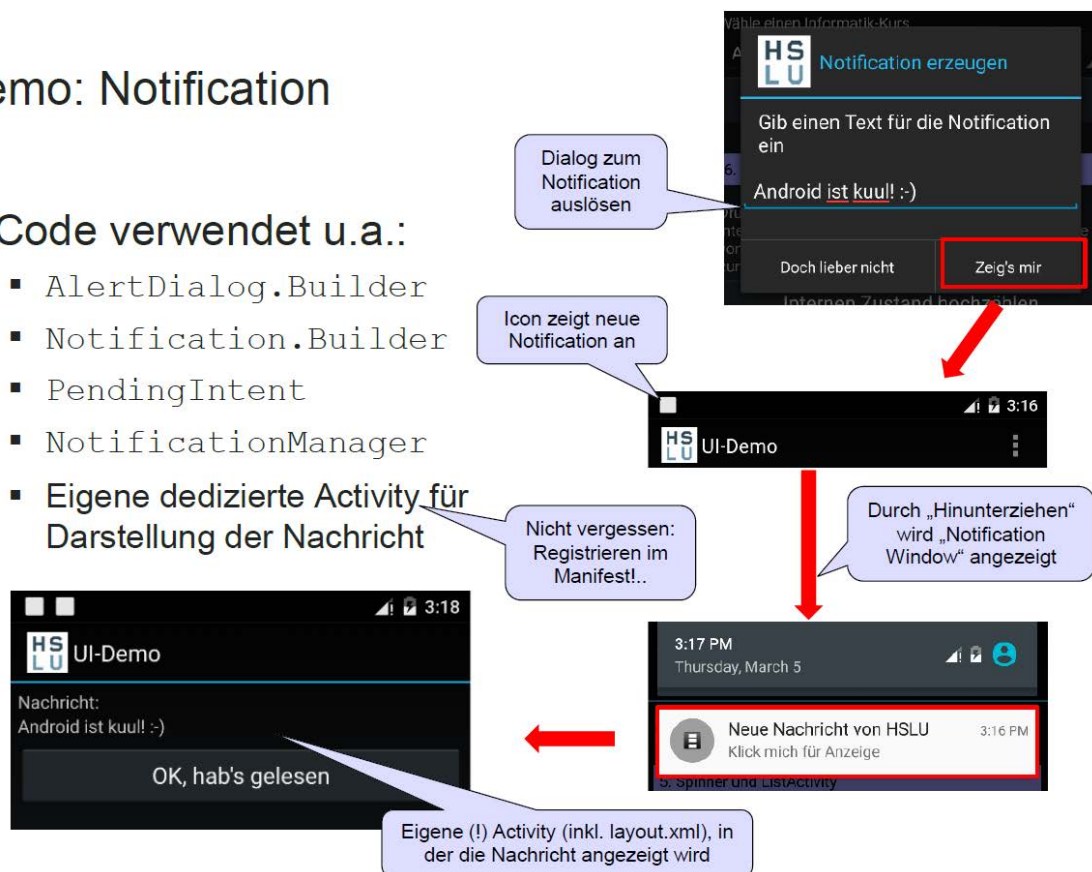


Abbildung 30: Übungs-Demo aus der Vorlesung SW02 - Notification

3 Android 3 - Persistenz & Content Providers

Persistenz: Daten über Laufzeit der App erhalten. Für lokale Persistenz gibt es drei Möglichkeiten:

- **Shared Preferences**
Key/Value-Paare, Verwendung für kleine Datenmengen
- **Dateisystem**
intern oder extern, in App-Sandbox (privat) oder auf SD-Karte (öffentlich), Verwendung für binäre/-grosse Dateien, Export
- **Datenbank (Room)**
SQLite + Object Relational Mapper (ORM), Verwendung für strukturierte Daten + Abfragen/Suche

3.1 (Shared) Preferences

- Jede Activity hat ein SharedPreferences-Profil, persistente Einstellungen für Activity oder Applikation
- Key-Value-Store (persistente Map)
- Preferences für **Activity**:
`Activity.getSharedPreferences(mode)`
Anwendungsfall: Activity-State persistent speichern
- Preferences für **Applikation**:
`PreferenceManager.getDefaultSharedPreferences(ctx)`
`Context.getSharedPreferences(name, mode)`
- Mögliche Datentypen für Preferences-Werte:
String, int, float, long, boolean, Set<String> (mit separaten Werten)

Lesen und Schreiben auf Preferences

- Mehrere Dateien pro Applikation möglich, Zugriff mit
`Activity.getSharedPreferences(name, mode)` (unterschiedliche Dateinamen)
oder auch über `getDefaultSharedPreferences(mode)`, die Applikation findet danach anhand der Preference-Benennungen die Einträge auch selber
- Lesen mit Methoden `SharedPreferences.getX()`
X steht für den Typ, also String, Int, Boolean, ...
- Schreiben immer mit dem Editor:
 1. `SharedPreferences.Editor editor = preferences.edit()`
 2. `editor.putX(...)`
 3. `editor.apply()` Persistierung der Änderungen
 - asynchrone Persistierung, blockiert die Methode nicht
 - für synchrone Persistierung: `editor.commit()`

Beispiel, um die Anzahl Aufrufe einer App über die Lebenszeit der App hinaus zu persistieren:

```
1 final SharedPreferences preferences = getPreferences(MODE_PRIVATE);
2 final int newResumeCount = preferences.getInt(COUNTER_KEY, 0) + 1;
3 final SharedPreferences.Editor editor = preferences.edit();
4 editor.putInt(COUNTER_KEY, newResumeCount);
5 editor.apply();
```

3.1.1 Darstellung User-Preferences

- Automatische Darstellung mit `PreferenceFragment`, eigener Editor für jeden Wertetyp
- `PreferenceFragment` schreibt/liest grundsätzlich in die `DefaultSharedPreferences`, kann aber auch für andere Preference-Stores konfiguriert werden

User-Präferenzen können in XML deklariert werden unter `res/xml` z.Bsp. als `preferences.xml`, wobei untersch. Präferenzen bspw. als `CheckBoxPreference`, `ListPreference` usw. erfasst werden. Daten können wie in diesem Beispiel aus den Array-Ressourcen bezogen werden:

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">
    <PreferenceCategory
        android:key="teaPrefs"
        android:title="Tee Präferenzen">

        <CheckBoxPreference
            android:key="teaWithSugar"
            android:persistent="true"
            android:summary="Soll der Tee gesüsst werden?"
            android:title="Gesüsster Tee?" />

        <ListPreference
            android:dependency="teaWithSugar"
            android:entries="@array/teaSweetener"
            android:entryValues="@array/teaSweetenerValues"
            android:key="teaSweetener"
            android:persistent="true"
            android:shouldDisableView="true"
            android:summary="Womit soll der Tee gesüsst werden?"
            android:title="Süsstoff" />

        <EditTextPreference
            android:key="teaPreferred"
            android:persistent="true"
            android:summary="z.B. "Lipton/Pfefferminztee""
            android:title="Bevorzugte Marke/Sorte" />

    </PreferenceCategory>
</PreferenceScreen>
```

Abbildung 31: Beispiel eines Präferenzen-XML

- Ohne `android:summary` würde die gewählte Preference angezeigt werden
- `android:dependency` deklariert eine Abhängigkeit zu einer anderen Preference, ist diese nicht gegeben kann die andere Preference nicht ausgewählt werden
- **Entries:** „Anzeigestring“, übersetzbar
EntryValues: „Werte“, nicht übersetzt, technischer Schlüssel

```
1 // Zur "Uebersetzung" von Values zu Entries (Beispiel)
2 public String getValueFromKey(String key) {
3     String[] keys = getResources().getStringArray(R.array.teaSweetenerValues);
4     String[] values = getResources().getStringArray(R.array.teaSweetener);
5     int i = 0;
6     while(i < keys.length) {
7         if(keys[i].equals(key)) {
8             return values[i];
9         }
10        i++;
11    }
12    return "";
13 }
```

3.1.2 PreferenceFragment

Ein PreferenceFragment kann in einer eigenen Activity (hier TeaPreferenceActivity) erstellt werden:

```
1 public class TeaPreferenceActivity extends Activity {
2
3     @Override
4     protected void onCreate(Bundle savedInstanceState) {
5         super.onCreate(savedInstanceState);
6         getFragmentManager().beginTransaction().replace(android.R.id.content,
7             new TeaPreferenceInitializer()).commit();
8     }
9
10    // PreferenceFragment als statische innere Klasse
11    public static final class TeaPreferenceInitializer extends PreferenceFragment
12    {
13        @Override
14        public void onCreate(final Bundle savedInstanceState) {
15            super.onCreate(savedInstanceState);
16            addPreferencesFromResource(R.xml.preferences);
17            // Referenz auf preferences.xml
18        }
19    }
```

3.1.3 Default-Präferenzen

Präferenzen können programmatisch auch wieder auf „Standard“-Werte oder auf festgelegte Werte gesetzt werden, für das Tee-Beispiel kann dies bspw. folgendermassen vorgenommen werden:

```
1 SharedPreferences teaPrefs = PreferenceManager.getDefaultSharedPreferences(this);
2 SharedPreferences.Editor editor = teaPrefs.edit();
3 editor.putString("teaPreferred", "Lipton/Pfefferminztee");
4 editor.putString("teaSweetener", "natural");
5 editor.putBoolean("teaWithSugar", true);
6 editor.apply();
```

3.2 Dateisystem

- **Einsatzbereiche**
 - Speichern/Laden von binären Dateien (Bilder, Musik, Video, Java-Objects, etc.)
 - Caching (Heruntergeladene Dateien)
 - Grosse Text-Dateien (Plain Text, Strukturierte Daten wie XML, JSON,)
- Teilen / Freigeben von erstelltem Inhalt (Externer Speicher wie SD-Karte)
- Dateien sind entweder
 - PRIVATE → ins Applikationsverzeichnis
(Zugriff für andere Apps nur über Content Provider möglich)
 - * `Context.getFilesDir()`
 - PUBLIC → auf die SD-Karte
 - * `Environment.getExternalStorageDirectory()`
`Environment.getExternalStorageState();`
- Für Zugriff auf SD-Karte muss die Permission im Manifest eingetragen werden! (siehe nachfolgend)

3.2.1 Exkurs: Permission-Model

- Vor gewissen Operationen müssen Apps die Berechtigung des Nutzers erhalten (Kontaktzugriff, Internet, SD-Karte, Kamera, SMS, etc.)
- Klasse: `android.Manifest.permission`
- Seit API 23 werden keine dangerous Permissions mehr gewährt, der Nutzer muss diese selber freigeben (Applikation fragt beim Nutzer nach), Permissions werden einzeln gewährt/abgelehnt.
Konsequenz: Apps müssen mit eingeschränkten Permissions umgehen können
- Arten von Permissions
 - *normal*
 - * Wird bei der Installation automatisch erlaubt
 - *dangerous*
 - * Muss von User erlaubt werden (kann wieder entzogen werden)
 - *signature*
 - * Wird automatisch erlaubt, wenn die App, welche die Permission definiert, vom gleichen Hersteller ist wie die App, welche die Permission beanträgt (sonst ist sie „dangerous“)
 - *signatureOrSystem*
 - * Wird automatisch erlaubt für Apps, welche im System-Image sind, sonst wie „signature“
- Permissions können gruppiert werden, User gibt Freigabe für alle Permissions in einer Gruppe (keine einzelnen Permissions), falls benötigt

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
    package="ch.hs1u.mobpro.persistence"
    xmlns:android="http://schemas.android.com/apk/res/android">

    <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
    <uses-permission android:name="android.permission.READ_SMS" />
    <uses-permission android:name="android.permission.WRITE_SMS" />

</manifest>
```

Abbildung 32: Erfassung von Permissions im Manifest

3.2.2 Exkurs ff: Runtime Permissions

```
public void loadExtFileWithPermission() {  
    int grant = checkSelfPermission(Manifest.permission.READ_EXTERNAL_STORAGE);  
    if (grant != PackageManager.PERMISSION_GRANTED) {  
        requestPermissions(new String[]{ Manifest.permission.READ_EXTERNAL_STORAGE }, 24);  
    } else {  
        // permission already granted  
        readFile();  
    }  
}
```

Abbildung 33: RuntimeCheck der Permissions

```
@Override  
public void onRequestPermissionsResult(int requestCode, String[] permissions, int[] grantResults) {  
    switch (requestCode) {  
        case 24: // load file  
            if (grantResults.length > 0 && grantResults[0] != PackageManager.PERMISSION_GRANTED) {  
                Toast.makeText(this, "Permission " + permissions[0] + " denied!", Toast.LENGTH_SHORT).show();  
            } else {  
                // permission was granted  
                loadFile();  
            }  
            break;  
    }  
}
```

Abbildung 34: Callback aus Permission-Abfrage

3.2.3 Exkurs ff: Persistenz mit Datei

Repetition zu Streams, Reader & Co.

- Stream: Byte-Datenstrom [28, 11, 200, 255, 2, 15, 33]
 - Auf File öffnen:
FileOutputStream, FileInputStream
- Stream kann in Zeichenstrom ['h', 'a', 'l', 'l', 'o'] umgewandelt werden
 - FileReader, FileWriter + „Buffered“-Versionen
- Immer schliessen!
stream.close() / reader.close()
- Nicht vergessen: try-catch-finally implementieren
- java.nio.file.Path: ist ab API 26 in Android verfügbar!

```
Writer writer = null;  
try {  
    writer = new BufferedWriter(new FileWriter(outFile));  
    writer.write(text);  
    return true;  
} catch (final IOException ex) {  
    // ...  
} finally {  
    Log.e("HSLU-MobPro-Persistenz", "Got a problem");  
    // ...  
}
```

Abbildung 35: Beispielcode zur Persistierung in einem Textfile

3.3 Datenbank (Room)

Android-DB **SQLite** ist bei Android fix integriert. Ein DB-Adapter ist die Verbindung zwischen Business-Objekten und einer Datenbank.

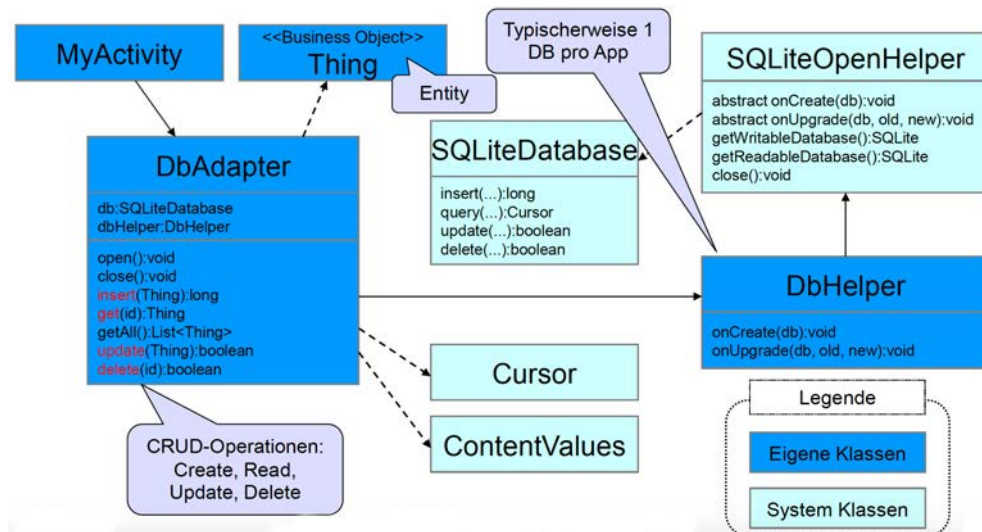


Abbildung 36: SQLite Framework

- Room ist ein Object Relational Mapper (ORM) für Android
 - Klassen werden auf relationale DB-Tabellen gemappt
 - Zugriff auf Datenbank wird abstrahiert
 - Typischerweise werden SQL-Statements durch Methodenaufrufe gekapselt
- Spezialfälle des Room ORM
 - Datenzugriff über DAO*: Queries werden als SQL-Statements in Annotationen definiert
 - Beziehungen zwischen Entitäten müssen manuell abgebildet werden (Performance!)
 - Nested Objects*: mehrere POJOs in einer Tabelle
 - Einschränkungen für Datenzugriffe, standardmässig nicht möglich im UI Thread
- Die drei Komponenten von Room
 - Database** Abstraktion der Datenverbindung
 - Entity** Repräsentation einer Tabelle in der relationalen DB
 - DAO** (Data Access Object) Enthält Methoden für Datenzugriff

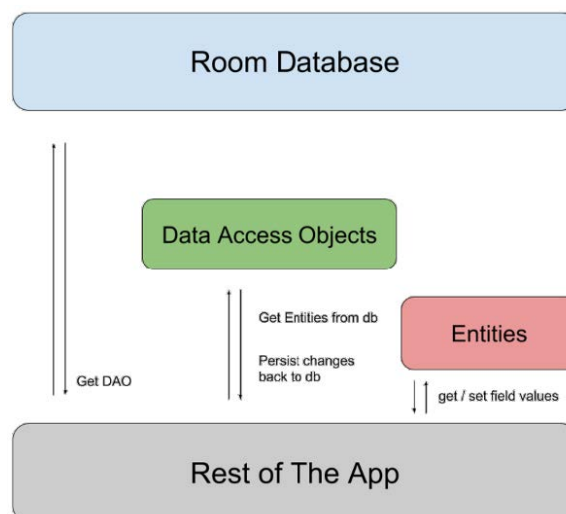


Abbildung 37: Komponenten von Room

3.3.1 Room - Code-Beispiele

```
1 @Entity // POJO mit Annotationen
2 public class User {
3     @PrimaryKey
4     public int uid;
5
6     @ColumnInfo(name = "first_name")
7     public String firstName;
8
9     @ColumnInfo(name = "last_name")
10    public String lastName;
11 }

```

```
1 @Dao // Datenzugriff ueber Annotationen (teilweise mit SQL-Queries)
2 public interface UserDao {
3     @Query("SELECT * FROM user")
4     List<User> getAll();
5
6     @Query("SELECT * FROM user WHERE uid IN (:userIds)")
7     List<User> loadAllByIds(int[] userIds);
8
9     @Insert
10    void insertAll(User... users);
11
12    @Delete
13    void delete(User user);
14 }

```

```
1 // Database: Subklasse von RoomDatabase, konfiguriert mit Database Annotation
2 @Database(entities = {User.class}, version = 1)
3 public abstract class AppDatabase extends RoomDatabase {
4     public abstract UserDao userDao();
5 }
6
7 // Zum Erzeugen einer Instanz der DB:
8 AppDatabase db = Room.databaseBuilder(
9     getApplicationContext(),
10    AppDatabase.class,
11    "database-name"
12 ).build();

```

3.3.2 Room - Daten mit Entitäten definieren

- Ein POJO mit @Entity Annotation
- Primärschlüssel (wird in jeder Entität benötigt)
 - @PrimaryKey für ein einzelnes Feld, optional mit autoGenerate Property
 - Für zusammengesetzte Primärschlüssel: primaryKeys Property in @Entity Annotation
- Falls bestimmte Felder nicht gespeichert werden sollen
 - @Ignore Annotation für ein einzelnes Feld
 - mit ignoredColumns Property in @Entity Annotation für mehrere Felder (v.a. von Superklassen)

```
1 // Code-Beispiel
2 // ACHTUNG: dieses Beispiel definiert mehrere Primary Keys und vermischte
   // Ansätze zum Ignorieren von Feldern zwecks Syntax-Demonstration!
3
4 @Entity(primaryKeys = {"firstName", "lastName"},
5         ignoredColumns = {"password", "otherField"})
6 public class User extends Party {
7     @PrimaryKey(autoGenerate = true)
8     public int id;
9
10    public String firstName;
11    public String lastName;
12
13    @Ignore
14    Bitmap picture;
15 }
```

3.3.3 Room - Beziehungen modellieren

Beachte: Room erlaubt aus Performanzgründen keine Objektreferenzierungen!

```
1 @Entity(foreignKeys = @ForeignKey(entity = User.class,
2                                   parentColumns = "id",
3                                   childColumns = "user_id"))
4 public class Book {
5     @PrimaryKey
6     public int bookId;
7
8     public String title;
9
10    @ColumnInfo(name = "user_id")
11    public int userId; // Feld-Typ: nur ID, nicht User
12 }
```

3.4 Mit DAOs auf Daten zugreifen

- DAOs enthalten Methoden für den abstrahierenden Datenbankzugriff
- Trägt zur *Separation of Concerns* bei und erhöht die Testbarkeit
→ DAOs können gemockt werden!
- DAOs als Interfaces oder abstrakte Klassen definieren
→ Room erzeugt passende Implementationen bei der Kompilierung
(*Typischerweise eine DAO-Klasse pro Entity, mit allen möglichen Operationen*)
- Zwei Möglichkeiten:
Convenience Queries **oder** `@Query` Annotation mit SQL-Statements

3.4.1 Convenience Queries

- Werden über Annotations für die jeweiligen Methoden definiert:
`@Insert`, `@Update`, `@Delete`
- Alle Parameter müssen Klassen mit einer `@Entity` Annotation
(oder Collections/Arrays) davon sein
- Rückgabewerte:
 - **Insert:** `long` bzw. `long[]` bzw. `List<Long>` (*liefert Row-IDs zurück*)
 - **Update / Delete:** `int` (*Anzahl modifizierte Tabelleneinträge*)

```
1 @Insert
2 public long[] insertUsersAndFriends(User user, List<User> friends);
3     // ID Rueckgabe, sonst void
4                                     // Parameter fuer Operation (Entities)
```

```
1 // Weitere Convenience Queries Beispiele
2
3 @Dao
4 public interface MyDao {
5
6     @Insert (onConflict = OnConflictStrategy.REPLACE)
7     public void insertUsers(User... users);
8
9     @Insert
10    public void insertBothUsers(User user1, User user2);
11
12    @Insert
13    public long[] insertUsersAndFriends(User user, List<User> friends);
14
15
16    @Update
17    public void updateUsers(User... users);
18
19    @Delete
20    public void deleteUsers(User... users);
21 }
```

3.4.2 Custom Queries mit @Query

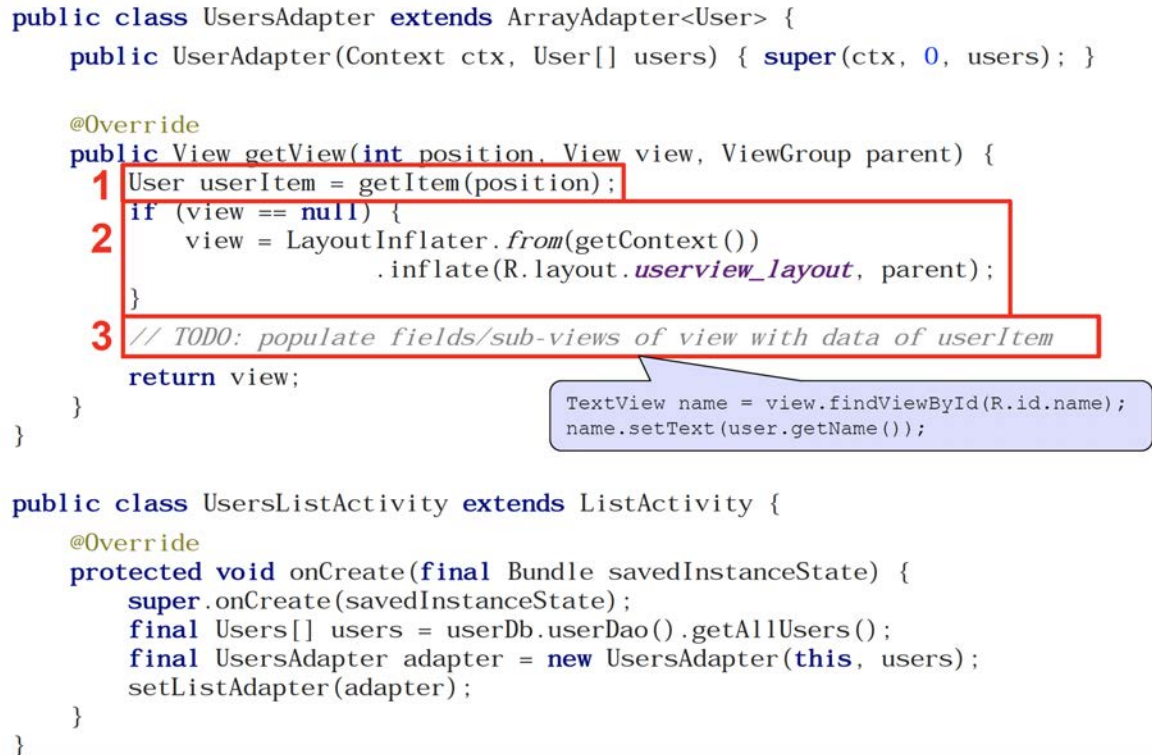
- Die @Query Annotation kann für Schreib- und Lesevorgänge genutzt werden
- Jede @Query wird zur Kompilierzeit überprüft
→ Kompilierfehler bei ungültigen Queries, keine Laufzeitfehler!
- Für eine @Query kann eine beliebige Anzahl (0..n) Parameter verwendet werden
- Wenn nicht ganze Objekte benötigt werden, können Ressourcen gespart werden durch die Verwendung von POJOs mit @ColumnInfo Annotationen

```
1 // Custom Queries Codebeispiele
2
3 @Dao
4 public interface MyDao {
5     @Query("SELECT * FROM user")
6     public User[] loadAllUsers();
7
8     @Query("SELECT * FROM user WHERE age > :minAge")
9     public User[] loadAllUsersOlderThan(int minAge);
10
11     @Query("SELECT first_name, last_name FROM user WHERE region IN (:regions)")
12     public List<NameTuple> loadUsersFromRegions(List<String> regions);
13 }
14
15 public class NameTuple {
16     @ColumnInfo(name = "first_name")
17     public String firstName;
18
19     @ColumnInfo(name = "last_name")
20     public String lastName;
21 }
```

3.5 DB-Einträge in einer Liste anzeigen

- Verschiedene Ansätze, je nach Umfang/Komplexität der Datensätze: ListView, RecyclerView, Kombination mit ViewModel und LiveData
- In jedem Fall werden spezifische Adapter benötigt, um die Daten auf Views zu mappen

```
public class UsersAdapter extends ArrayAdapter<User> {  
    public UsersAdapter(Context ctx, User[] users) { super(ctx, 0, users); }  
  
    @Override  
    public View getView(int position, View view, ViewGroup parent) {  
        1 User userItem = getItem(position);  
        2 if (view == null) {  
            view = LayoutInflater.from(getContext())  
                .inflate(R.layout.userview_layout, parent);  
        }  
        3 // TODO: populate fields/sub-views of view with data of userItem  
        return view;  
    }  
}  
  
public class UsersListActivity extends ListActivity {  
    @Override  
    protected void onCreate(final Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        final Users[] users = userDao().getAllUsers();  
        final UsersAdapter adapter = new UsersAdapter(this, users);  
        setListAdapter(adapter);  
    }  
}
```



```
TextView name = view.findViewById(R.id.name);  
name.setText(user.getName());
```

Abbildung 38: Codebeispiel für das Darstellen von DB-Einträgen in einer Liste

3.6 Content Providers

- Content Provider stellen für andere Applikationen Daten bereit
- Die Daten stammen aus einer gekapselten DB **oder** aus dem privaten Dateisystem **oder** werden on-the-fly erzeugt
- Zugriff auf die Daten über URI (Uniform Ressource ID), Beispiel siehe in Abbildung 39
- Zwei Arten von URIs
 - Pfad (Bezeichnete Datenmenge, vgl. Verzeichnit mit Daten)
 - Item (Einzelnes Datenelement, vgl. einzelne Datei)

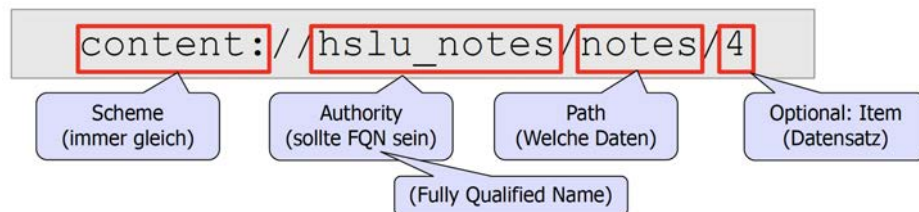


Abbildung 39: Aufbau eines URI

3.6.1 Standard Content Providers

- Im Android-System gibt es bereits einige Content Providers, die genutzt werden können
 - Kontakte: Namen, Telefon-Nummern, Emails, Adressen, etc.
 - SMS/MMS: Erhaltene/Gesendete/Drafts SMS/MMS
 - Media Store: Auf Gerät gespeicherte Audio-, Video-, Bilder-Daten
 - Settings: Einstellungen für das Gerät
 - Kalender: Kalender, Events, Erinnerungen, Teilnehmer, etc.
- Daten sind meist in mehreren Tabellen abgelegt

3.7 Exkurs : REST-ful Webservices

- Webservices auf der Basis von HTTP
- Grundidee (in purer Form)
 - URL einer Ressourcensammlung (*http://directory.com/contacts*) oder URL einzelner Ressource (*http://directory.com/contacts/17*)
 - HTTP-Methode = Operation auf Daten (GET, PUT, POST, DELETE)
 - Antwort-Datenformat = XML, JSON, ...

Resource	GET	PUT	POST	DELETE
Collection URI, such <i>http://directory.com/contacts/</i>	List the URIs and perhaps other details of the collection's members.	Replace the entire collection with another collection.	Create a new entry in the collection. The new entry's URL is usually returned by the operation.	Delete the entire collection.
Element URI, such as <i>http://directory.com/contacts/17</i>	Retrieve a representation of the addressed collection member, expressed in an appropriate media type.	Replace the addressed member of the collection, or if it doesn't exist, create it.	Treat the addressed member as a collection in its own right and create a new entry in it.	Delete the addressed member of the collection.

Abbildung 40: Beispiel - Representational State Transfer

3.8 Content Resolver & Content Provider

- Zugriff auf einen Content Provider erfolgt über einen **Content Resolver**
`Context.getContentResolver()`
 - Bietet DB-Methoden und Zugriff auf Content via Streams
 - * CRUD: `insert()` / `query()` / `update()` / `delete()`
 - * `openInputStream(uri)` / `openOutputStream(uri)`
 - Ein Content Resolver ist ein Proxy, der...
 - * ...URI auflöst und zuständigen Content Provider sucht / findet
 - * ...Interprozess-Kommunikation behandelt (aufrufende App ist meist in einem anderen Package als der aufgerufene Content Provider)
- Unter Umständen müssen die Permissions noch gesetzt werden (im Manifest)

3.8.1 Zugriff auf Daten über Content Resolver & Query

```

1 Cursor cursor = getContentResolver().query(
2     contentUri,        // The content URI of the table
3     projection,        // The columns to return for each row
4     selectionClause,    // Selection criteria
5     selectionArgs,      // Selection criteria
6     sortOrder);        // The sort order for the returned row

```

Content Provider Query	SQL SELECT Query	Notes
contentUri	FROM table_name	contentUri maps to the table in the provider named table_name.
projection	Col, col, col,...	projection is an array of columns that should be included for each row retrieved.
selection	WHERE col = value	selection specifies the criteria for selecting rows.
selectionArgs	(No exact equivalent. Selection arguments replace ? placeholders in the selection clause.)	-
sortOrder	ORDER BY col,col,...	sortOrder specifies the order in which rows appear in the returned Cursor.

Abbildung 41: Vergleich: ContentProvider Query und SQL Query Parameter

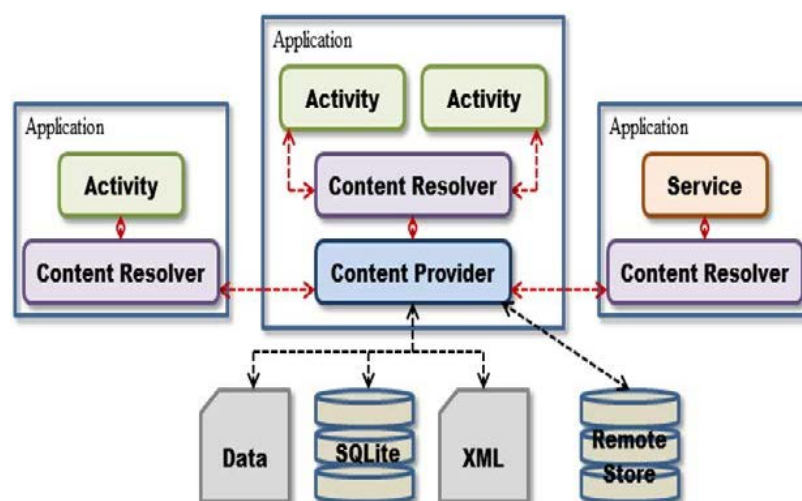


Abbildung 42: Content Provider - Anwendung (Data: Dateisystem, XML: Preferences)

- SMS des Systems sind über den Content Provider zugänglich
(Benötigt Permission für SMS, diese testen und ggf. beantragen):
 - android.provider.Telephony.Sms
(„Sub-Providers“ für Sent, Inbox, Draft, etc.)
 - Im Package android.provider.* finden wir
„Contract Klasse“ Telephone.Sms mit
Hilfsklassen BaseColumns und Telephony.TextBasedSmsColumns
(Hier findet man Content-URI und Spalten-Namen für Projections)

```
public void showSmsList(final View view) {
    final Cursor cursor = getContentResolver().query(
        Telephony.Sms.Inbox.CONTENT_URI, // content uri
        new String[]{ Telephony.Sms.Inbox._ID, Telephony.Sms.Inbox.BODY }, // projection
        null, null, null); // selection, selection args, sort order

    new AlertDialog.Builder(this)
        .setTitle("SMS in Inbox")
        .setCursor(cursor, null, Telephony.TextBasedSmsColumns.BODY)
        .setNeutralButton("Ok", null)
        .create()
        .show();
}
```

Abbildung 43: Anwendungsbeispiel - Alle SMS mit Text anzeigen

Jeder Content Provider bietet eine eigene Standard-API, in der Android Dokumentation sind die Zugriffe auf Kontakte und Kalender gut dokumentiert (da dies eher komplizierte Modelle sind). Einstiegspunkt für die meisten Provider: android.provider.*

3.9 Eigener Content Provider

- Um eigenen Content Provider zu schreiben, muss die eigene Klasse von der abstrakten Klasse android.content.ContentProvider ableiten
- Wird bei App-Start hochgefahren und bleibt aktiv, in onCreate() kann eine Initialisierung vorgenommen werden (einzige Lifecycle-Methode)
- CRUD-Methoden: query, insert, update, delete (muss nicht alle implementieren)
(Möglichkeit, einen read-only Content Provider anzulegen)

Demo: Content Provider für Notizen

- Dialog zeigt Notizen an
- Nur für internen Gebrauch
 - exported=false
- NotesProvider: Konstanten definiert in NotesContract
- Aufrufender Code in Activity:

```
public void readNotesFromContentProviderOnClick(final View view) {
    final Cursor cursor = getContentResolver().query(
        Uri.parse(NotesContract.CONTENT_URI), NotesContract.PROJECTION,
        null, // SELECT *
        null, NotesContract.COLUMN_TITLE);

    new AlertDialog.Builder(this).setTitle("Notizen via ContentProvider")
        .setCursor(cursor, null, NotesContract.COLUMN_TITLE)
        .setPositiveButton("Toll!", null)
        .setNegativeButton("Abbruch", null).create().show();
}
```

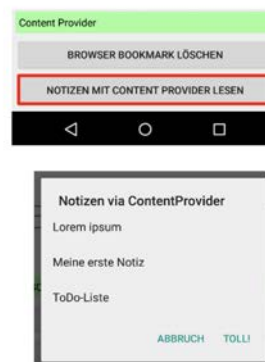


Abbildung 44: Anwendungsbeispiel - Content Provider für Notizen

4 Android 4 - Kommunikation & Nebenläufigkeit

4.1 Nebenläufigkeit

4.1.1 Android und der Main-Thread

- Eine Applikation baut ihr UI nur auf einem Thread, dem **main-Thread** auf
→ blockiert man den main-Thread, friert das ganze UI ein
- UI-Komponenten sind nicht Thread-safe
→ UI-Zugriff nur aus main-Thread, sonst Exception
- Netzwerk- und andere Methoden können lange dauern und sind blockierend, werden diese auf dem main-Thread aufgerufen, wird er blockiert und es werden keine UI-Events mehr aufgerufen
Bsp. `URLConnection.connect()`, `Bitmap.resize()`, `Database.open()`, ...

```
public void freeze7Seconds(View view) {  
    try {  
        Thread.sleep(WAITING_TIME_MILLIS);  
    } catch (InterruptedException e) {  
        // ignore  
    }  
}
```

Abbildung 45: Einfaches Blockieren der App mittels `Thread.sleep(long time)`

4.1.2 Android-Überwachung - ANR (Application Not Responding)

- Android überwacht Ansprechbarkeit von Apps nach gewissen Kriterien
 - Keine Reaktion auf Input-Event innert 5 Sekunden
 - Broadcast-Receiver nicht fertig innert 10 Sekunden
- Möglicher Effekt ist ein ANR-Dialog („App reagiert nicht“)
- System-Mechanismus zum Stoppen „böser“ Apps
- Fragen, die man sich stellt, um den main-Thread zu entlasten:
 - Ist eine Hintergrundaufgabe aufschiebbar?
 - Hat ein Task Auswirkungen auf das UI?
 - Wartet der User auf ein Resultat?
- Herausforderungen dabei:
 - Das Resultat am Ende auf dem UI-Thread darstellen
 - Ist das UI noch da?
- Wenn eine Aktion nicht aufschiebbar ist:
 1. **Klasse AsyncTask**
Konstrukt zur Auslagerung zeitintensiver Aufgaben auf Background-Task, für die meisten Fälle ausreichend
 2. **Eigene Thread Instanzen**
Standardimplementierung von Nebenläufigkeit in Java, kann komplex werden: Synchronisierung, Deadlocks, ...
 3. **Foreground Service**
Hintergrundaktionen, die vom Nutzer bemerkt werden z.Bsp. Musik-Player
- Zurückgelangen zum UI-Thread:
 1. Klasse `AsyncTask`:
Spezielle Methoden auf Main-Thread, bspw. `onProgressUpdate`, `onPostExecute`, ...
 2. Eigener Thread, zwei Möglichkeiten:
 - `Activity.runOnUiThread(Runnable action)`
 - `View.post(Runnable action)`
 - Klasse `android.os.Handler` → nutzt Message Queue von Thread

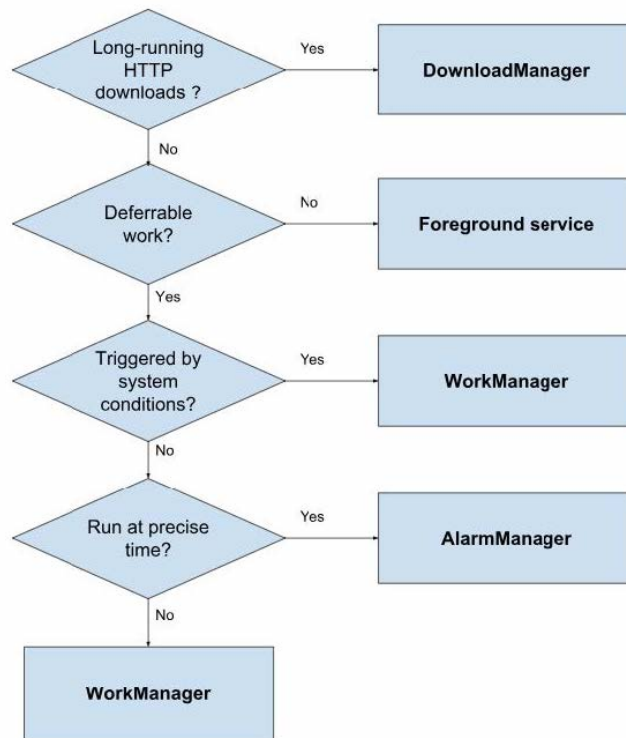


Abbildung 46: Verschiedene Arten für Hintergrundaufgaben

Langandauernde Operationen Android lässt gewisse Operationen auf main-Thread gar nicht erst zu! Bsp. Netzwerk-API, bspw. Aufruf von `URLConnection.connect()` führt zu einer

NetworkOnMainThreadException

Grund dafür ist, dass Netzwerk-Kommunikation lange dauern kann, Netzwerk-Calls nie auf UI-Thread ausführen, **AsyncTask oder eigenen Thread verwenden** (und UI-Aktualisierungen in Methoden auf dem main-Thread!)

4.2 Nebenläufigkeit: AsyncTask

```
class MyAsyncTask extends AsyncTask<Params, Progress, Result> {
    protected Result doInBackground(Params... params) // muss implementiert werden
    protected void onProgressUpdate(Progress... progress) // optional
    protected void onPostExecute(Result result) // optional
    public void execute(Params... params) // starte Task - nur einmal möglich!
}
```

Abbildung 47: MyAsyncTask Klasse

- 3 Typ-Parameter
 - **Params**: Typ der Input-Elemente,
Bsp. URL (Links auf Textfiles,...)
 - **Progress**: Typ der Zwischenresultate,
Bsp. String (Void falls nicht genutzt) (Heruntergeladener Titel, ...)
 - **Result**: Typ des Resultats,
Bsp. Integer (Void falls nicht genutzt) (Anzahl heruntergeladener Titel, ...)
- 3 wichtige Methoden
 - `doInBackground(Params...)`: Lange andauernd (Worker-Thread)
 - `onProgressUpdate(Progress...)`: Zwischenresultat verarbeiten (UI-Thread)
 - `onPostExecute(Result)`: Resultat verarbeiten (UI-Thread)
- Schlussfolgerungen:
 1. AsyncTasks werden standardmässig (seriell) durch einen einzelnen Worker-Thread ausgeführt.
 2. Thread-Pool (parallele Ausführung) durch Angabe Executor möglich.
 3. Implementierungs-Details (ggf. auch relevante) können sich ändern.

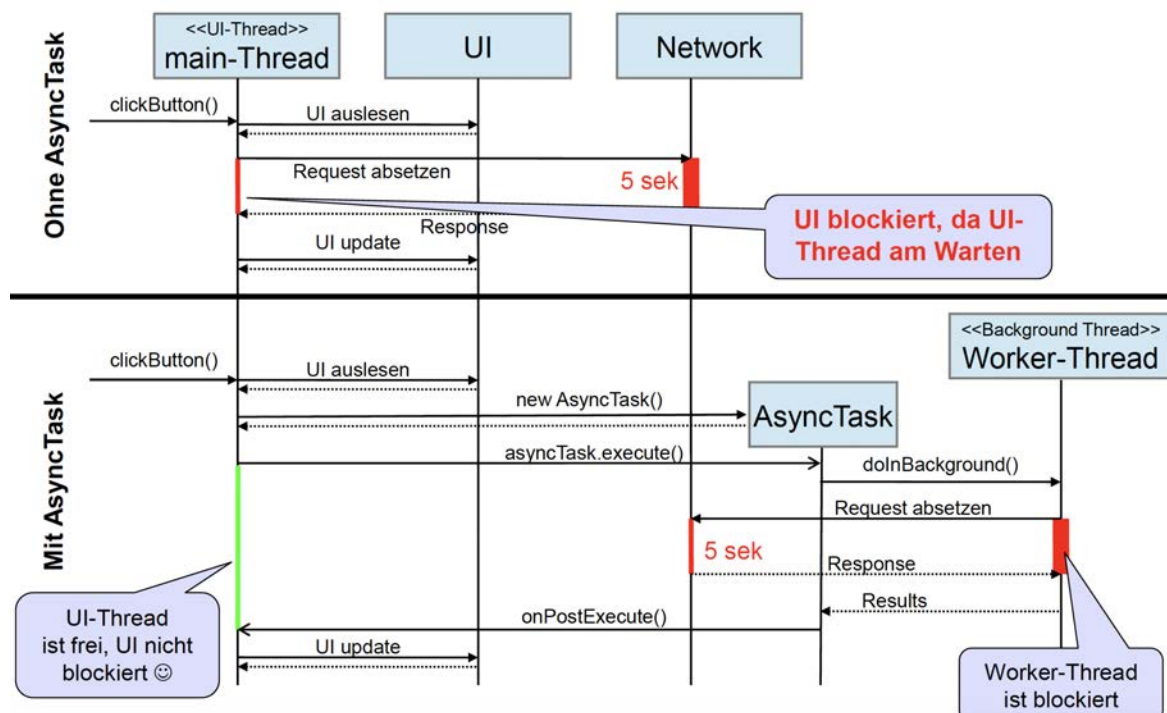


Abbildung 48: Ablauf mit und ohne AsyncTask

4.2.1 Einige Demos aus dem Unterricht

Demo: 7 Sek warten mit eigenem AsyncTask

Parameter-Typ Zwischenresultat-Typ Resultat-Typ

```
public class AsyncDemoTask extends AsyncTask<Integer, Void, String> {
```

- Hintergrund-Thread 7 Sek blockieren
 - In `AsyncTask.doInBackground(...)`
- Argument vom Typ `Integer`
 - D.h. Übergabe von `Integer[]` an `doInBackground(...)`
- Resultat vom Typ `String`
 - D.h. `doInBackground(...)` gibt `String` zurück
 - Ausgabe in Toast bei Task-Ende
- Nur ein `AsyncTask` soll aktiv sein
 - Ein bisschen Logik dafür selber implementieren...

AsyncTask starten

[AsyncTask läuft...]

UI friert nicht ein! 😊

AsyncTask starten
Resultat = 'Die Parameter waren: 77, 444, 2000, -23, 111'.

AsyncDemoTask läuft schon!

Demo: AsyncTask mit Zwischenresultaten (Progress)

- Idee: Mehrere Textdateien von Server holen:
 - Zwischenresultate in Toast:
 - Benutzen Methoden
 - `publishProgress(...)`
 - `onProgressUpdate(...)`
 - Am Ende Dialog mit allen Texten (Filmtiteln):
 - `onPostExecute(...)`
- Daten-URL: `http://wherever.ch/hslu/titleX.txt`
 - für X von 0..4

AsyncMultiTask starten

Neuer Titel: 'Pulp Fiction'.

HS LU Film Titel

Reservoir Dogs

Pulp Fiction

Jackie Brown

Kill Bill

Django Unchained

Aha...

Code-Ausschnitte "Demo mit Zwischenresultaten"

```

public class MultiAsyncTask extends AsyncTask<URL, String, Void> {

    @Override
    protected Void doInBackground(URL... urls) {
        try {
            for (URL url : urls) {
                InputStream in = openHttpConnection(url);
                String text = readText(in);
                in.close();
                Thread.sleep(WAIT_TIME_MILLIS);
                publishProgress(text);
            }
        }
    }

    @Override
    protected void onProgressUpdate(String... values) {
        super.onProgressUpdate(values);
    }

    @Override
    protected void onPostExecute(Void result) {
        AlertDialog.Builder dialogBuilder = new AlertDialog.Builder(MainActivity);
        dialogBuilder.setTitle("Film Titel")
    }
}

```

Annotations and Callouts:

- Zwischenresultat-Typ**: Points to `String` in the generic type `URL, String, Void`.
- Resultat-Typ**: Points to `Void` in the generic type `URL, String, Void`.
- Parameter-Typ**: Points to `URL... urls` in the `doInBackground` method signature.
- Läuft im Hintergrund auf separatem Thread**: Points to the `doInBackground` method.
- Ruft onProgressUpdate(...) auf**: Points to the `publishProgress(text);` line.
- Zwischenresultat-Typ**: Points to `String... values` in the `onProgressUpdate` method signature.
- Läuft auf main-Thread = UI-Thread**: Points to the `onProgressUpdate` method.
- Resultat-Typ**: Points to `Void result` in the `onPostExecute` method signature.

AsyncTask: Methoden & Threads...

...hier leistet der Debugger gute Dienste! ☺

Annotations and Callouts for Thread Debugging:

- MainActivity läuft im main-Thread (= UI-Thread)**: Points to the `"main" @3,563 in group "main": WAIT` thread in the first screenshot.
- AsyncTask.doInBackground() läuft in eigenem Thread**: Points to the `"AsyncTask #1" @3,588 in group "main": WAIT` thread in the second screenshot.
- AsyncTask.onProgressUpdate() läuft im main-Thread**: Points to the `"main" @3,563 in group "main": WAIT` thread in the third screenshot.
- AsyncTask.onPostExecute() läuft im main-Thread**: Points to the `"main" @3,563 in group "main": WAIT` thread in the fourth screenshot.

The screenshots show the Android Studio interface with the **Threads** tab selected, displaying the execution flow and thread states for the application.

4.3 Nebenläufigkeit: Threads

Repetition `java.lang.Thread` implements `Runnable`

- Muss `Runnable` gesetzt haben, übergeben im Konstruktor, oder selber `run()` implementieren
- Wichtigste Methoden:
 - `run()`
 - `start()`
 - `sleep(long)`
 - `isAlive()`
- Interface `java.lang.Runnable` (oder Lambda)
 - Eine Methode: `run()`
 - Implementierung des relevanten nebenläufigen Codes

Verwendung von Threads

In der `run`-Methode darf nicht auf die View-Elemente zugegriffen werden

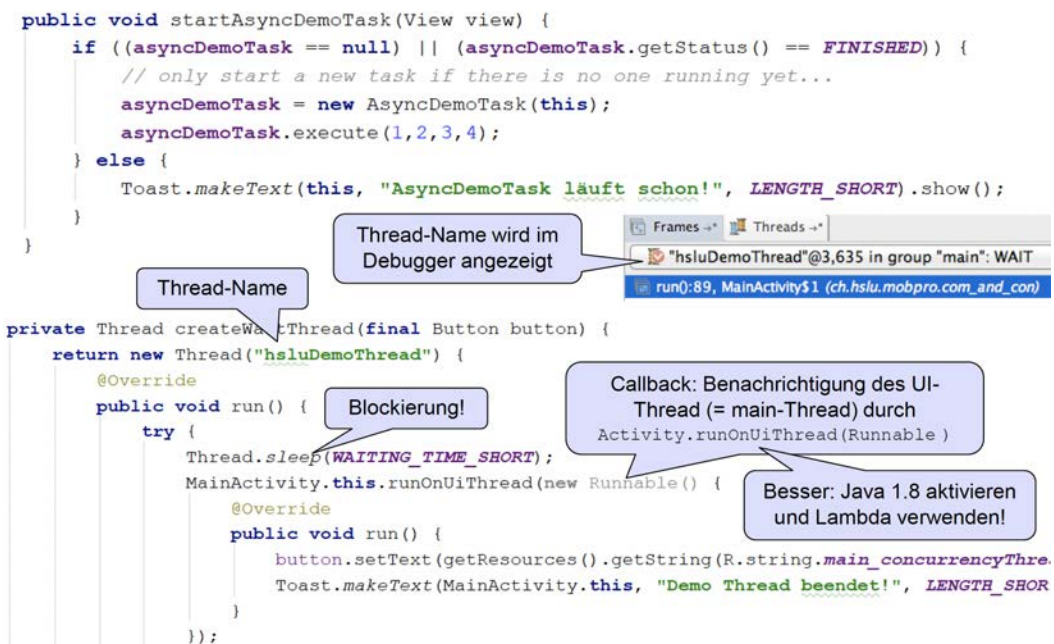


Abbildung 49: Verwendung von Threads

4.4 (Backend-) Kommunikation über HTTP

Apps können mit Server im Hintergrund kommunizieren (hält Daten, stellt Business-Logik bereit, authentisiert User etc.). Kommunikation findet i.d.R. über (REST) HTTP-API im Datenformat JSON (seltener XML) statt.

- HTTP: Hyper Text Transfer Protocol
 - Zustandsloses Kommunikationsprotokoll
- Transport über TCP / IP
- Request / Response Muster (Anfrage/Antwort):
 - GET, PUT, POST, DELETE
- Nachrichten bestehen aus Header & Body
 - 0 .. n Headers: Key-Value Paare
 - Body (Content): beliebig, typischerweise Text
- Mit jeder Antwort liefert Server einen Antwortcode (Bsp. 200 = OK)

4.4.1 HTTP-Requests absetzen

- **Veraltet:**
URL und URLConnection: Standard-Java-Klassen, erlauben das Absetzen von HTTP-Requests, mühsame Verwendung, veraltete API
- **Besser:**
HTTP-Client-Library verwenden, „headless Browser“
Empfohlen: OkHttpClient, Gradle Dependencies:
com.squareup.okhttp3:okhttp:3.13.1
com.squareup.okhttp3:logging-interceptor:3.12.1

Verwendung OkHttpClient (Text)

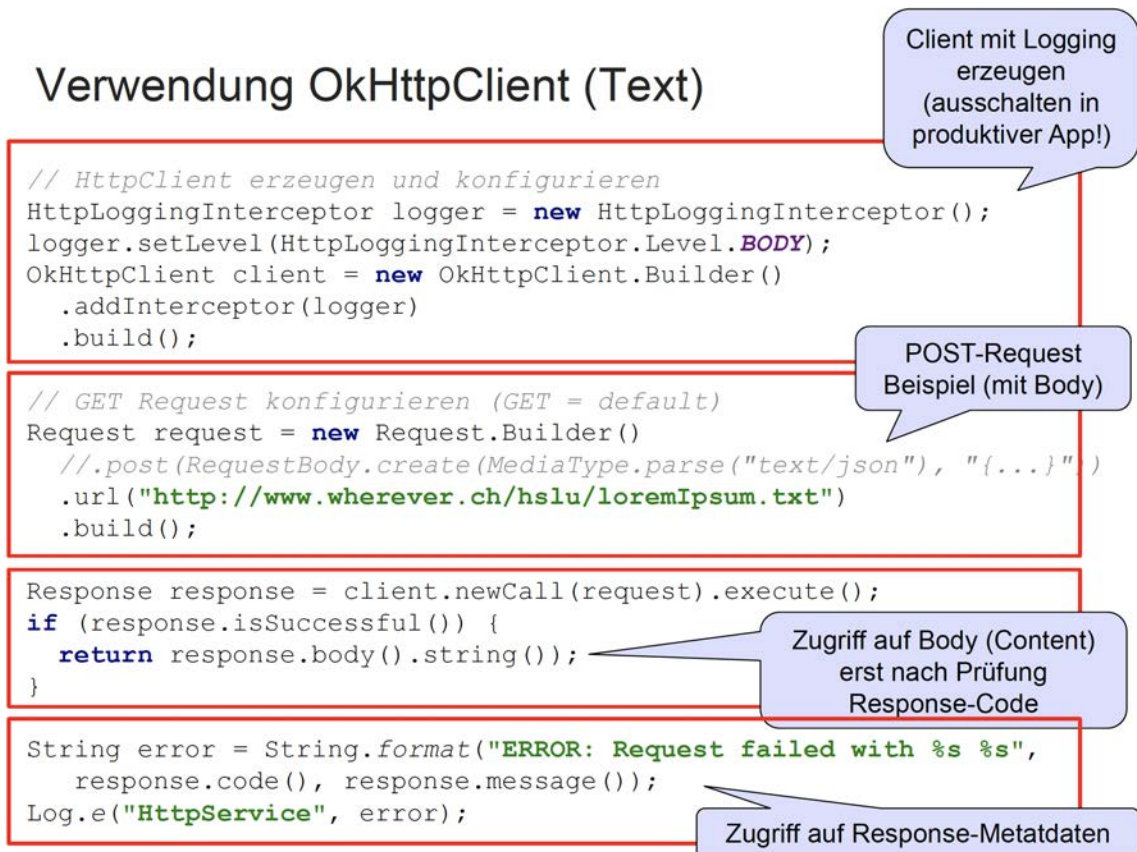


Abbildung 50: Beispiel der Verwendung von OkHttpClient

- Unverschlüsselte Kommunikation über HTTP seit Android 9 (API 28) unterbunden, kann im Manifest explizit erlaubt werden:
`android:usesCleartextTraffic="true"`
- Für den Internetzugriff (und Netzwerkstatus) muss die Berechtigung vorliegen bzw. deklariert werden, dies kann im Manifest geschehen:
`uses-permission android:name="android.permission.INTERNET"`
`uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"`

Demo für lokales Testen der Backend-Kommunikation und Umwandlung von Binärdaten können in den MOBPRO-Folien „*MobPro_Android_4*“ eingesehen werden.

4.5 JSON-Webservices mit Retrofit konsumieren

Möglichst immer REST-Semantik (HTTP-Methoden für gewünschte Operation) verwenden.

- **REST-ful Webservices**

Webservice auf Basis von HTTP

- Grundidee:

- Base-URL: Ressourcensammlung / einzelne Ressource
(*http://directory.com/contacts/17*)
- HTTP-Methode: Operation auf Daten
(*GET, PUT, POST, DELETE*)
- Antwort-Datenformat: XML, JSON, ...

Resource	GET	PUT	POST	DELETE
Collection URI, such as <i>http://directory.com/contacts/</i>	List the URIs and perhaps other details of the collection's members.	Replace the entire collection with another collection.	Create a new entry in the collection. The new entry's URL is usually returned by the operation.	Delete the entire collection.
Element URI, such as <i>http://directory.com/contacts/17</i>	Retrieve a representation of the addressed collection member, expressed in an appropriate media type.	Replace the addressed member of the collection, or if it doesn't exist, create it.	Treat the addressed member as a collection in its own right and create a new entry in it.	Delete the addressed member of the collection.

Abbildung 51: Beispiel für RESTful Webservice

4.5.1 JSON Parsing mit GSON

Gson ist ein JSON to Java Mapper, welcher JSON-Strukturen auf äquivalente Java-Klassen abbildet (ähnlich ORM bei Room).

Beispiel:

```
1 String url = "http://www.nactem.ac.uk/.../dictionary.py?sf=HTTP";
2 OkHttpClient client = new OkHttpClient();
3 Request request = new Request.Builder().url(url).build();
4 Response response = client.newCall(request).execute();
5 String json = response.body().string();
6
7 Gson gson = new Gson();
8 Type listType = new TypeToken<List<AcronymDef>>(){}.getType();
9 List<AcronymDef> definitions = new Gson().fromJson(json, listType);
```

4.5.2 Retrofit

Mit Retrofit möchte man das Backend als Interface abstrahieren, um HTTP-Calls mit Java-Calls zu ersetzen, wie folgt:

```
1 public interface AcronymService {
2
3     @GET("dictionary.py")
4     Call<List<AcronymDef>> getDefinitionsOf(@Query("sf") String sf);
5
6 }
```

Retrofit basiert auf OkHttp, diese Library ist also automatisch vorhanden. Man kann verwendete OkHttpClient-Instanzen auch konfigurieren, bevor man sie auf die Retrofit-Factory setzt.

Gradle Dependencies:

```
com.squareup.retrofit2:retrofit:2.5.0
```

```
com.squareup.retrofit2:converter-gson:2.5.0 (JSON Mapper)
```


Retrofit Konfiguration und Aufruf

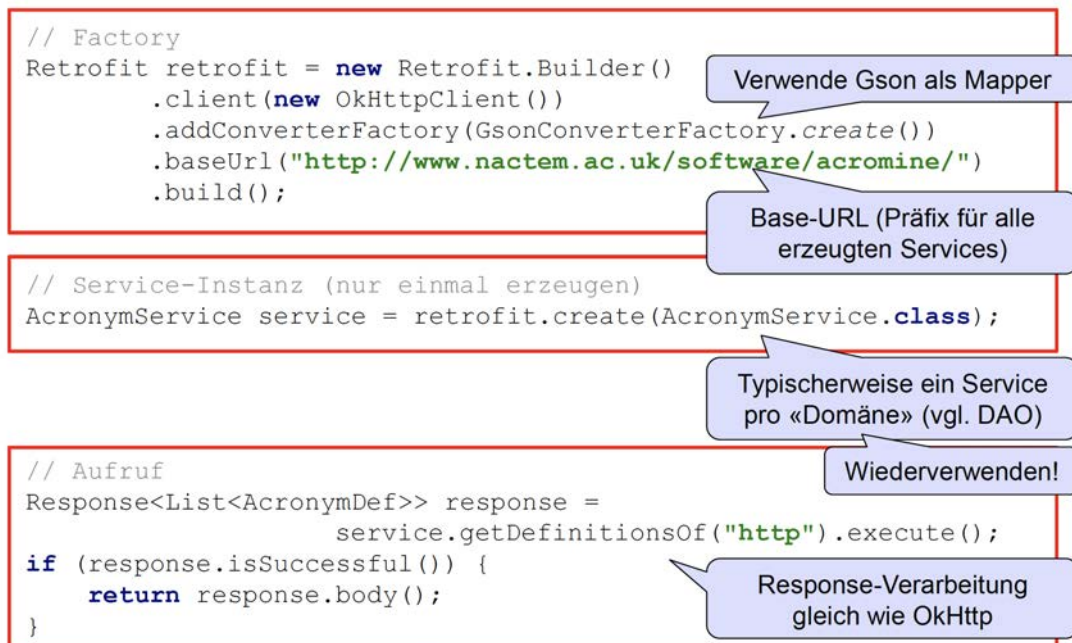
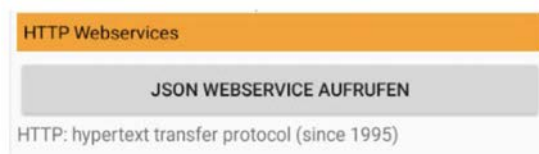


Abbildung 52: Konfiguration und Aufruf von Retrofit

Demo: JSON-Service mit Retrofit konsumieren (Ü4)

- Service-URL:
`http://www.nactem.ac.uk/software/acromine/dictionary.py`
- Argument:
`sf=HTTP`
- Logging in Konsole:



```
2019-03-11 18:04:50.197 D/OkHttp: --> GET http://www.nactem.ac.uk/software/acromine/dictionary.py?sf=HTTP
2019-03-11 18:04:50.198 D/OkHttp: --> END GET
2019-03-11 18:04:51.016 D/OkHttp: <-- 200 OK http://www.nactem.ac.uk/software/acromine/dictionary.py?sf=HTTP (818ms)
2019-03-11 18:04:51.017 D/OkHttp: Date: Mon, 11 Mar 2019 17:04:53 GMT
2019-03-11 18:04:51.017 D/OkHttp: Server: Apache/2.2.15 (Scientific Linux)
2019-03-11 18:04:51.017 D/OkHttp: Connection: close
2019-03-11 18:04:51.018 D/OkHttp: Transfer-Encoding: chunked
2019-03-11 18:04:51.018 D/OkHttp: Content-Type: text/plain; charset=UTF-8
2019-03-11 18:04:51.047 D/OkHttp: [{"sf": "HTTP", "lfs": [{"lf": "hypertext transfer protocol", "freq": 6, "since": 1995, "vars": [{"lf": "Hypertext Transfer Protocol", "freq": 3, "since": 1996}, {"lf": "hypertext transfer protocol", "freq": 3, "since": 1995}]}]}]
2019-03-11 18:04:51.047 D/OkHttp: <-- END HTTP (231-byte body)
```

Abbildung 53: Demo: JSON-Service mit Retrofit konsumieren

5 Android 5 - Services & Broadcast Receiver

5.1 Service-Komponente

- Services wurden ursprünglich zur Kapselung und Erledigung von Hintergrundaufgaben eingeführt. Seit API 26 (Android 8 Oreo) stark eingeschränkt → System Performance Issues (zu viele Services in zu vielen Apps)
- **Heute:** Nur als Spezialfall „Foreground Service“ und zum Export der App-Logik (Sicherheitsrisiko) empfohlen (Bsp. Musikplayer)
- Für frühere Anwendungsbereiche: **WorkManager** empfohlen (Bsp. Location Update, Background Sync etc.)

5.1.1 Service-Konzept

- Was **kann** ein Service?
 - Dem System mitteilen, dass eine Arbeit im Hintergrund ausgeführt werden soll
`startService()` - Auftrag für Service erteilen
 - Gewisse Funktionalität (API) exportieren und anderen Apps anbieten
`bindService()` - öffnet stehende Verbindung für Kommunikation mit Services
- Was kann ein Service **nicht**?
 - Kein separater Worker-Thread (per se)
 - Kein eigener Prozess (nur wenn als solcher definiert)
- Ein Service **kann und sollte** einen eigenen Thread für langandauernde Operationen starten

5.1.2 Lebensarten & Lifecycle-Methods eines Service

- **Ungebundener Service:** Service verbleibt im Zustand RUNNING bis er explizit beendet wird
Aufruf durch `startForegroundService(...)`
 - `onCreate()`: Bei Erzeugung
 - `onStartCommand()`: Auftragsbehandlung
 - `onDestroy()`: Bei Beendung (durch Service/App/System)
- **Gebundener Service:** Service verbleibt nur so lange im Zustand RUNNING wie Bindings existieren
Aufruf durch `bindService()`
 - `onCreate()`: Bei Erzeugung
 - `onBind()`: wenn Komponente Verbindung herstellt
 - `onUnbind()`: wenn Komponente Verbindung beendet
 - `onDestroy()`: Bei Beendung (durch Service/App/System)

Auf der nachfolgenden Seite sind die Lifecycle-Methoden eines Service übersichtlich dargestellt.

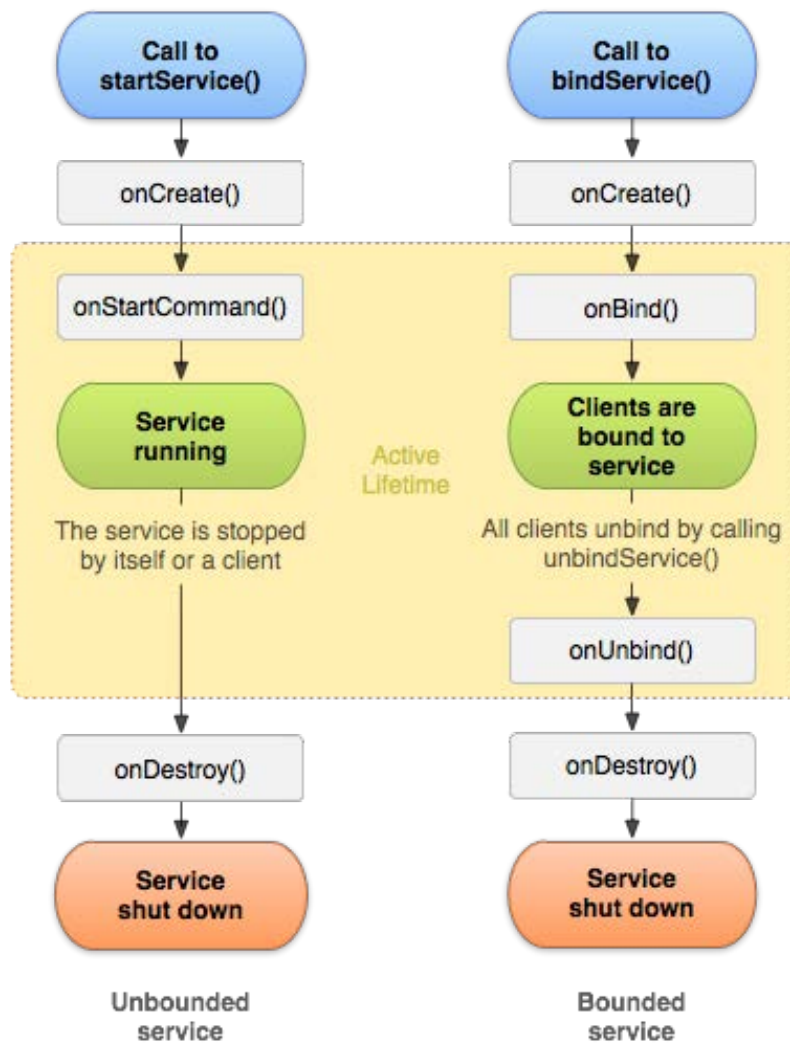


Abbildung 54: Lifecycle-Methoden von Services
Ab API 26 ist `startService()` → `startForegroundService()`

5.1.3 Demo-Code eines Foreground-MusicPlayer-Service

- Hintergrundarbeit, die für den Benutzer wahrnehmbar ist
- Zeigt eine Notification, während der Service läuft
- Benötigt Permission: `FOREGROUND_SERVICE`
- Interaktion / Steuerung des Service oft über Binding

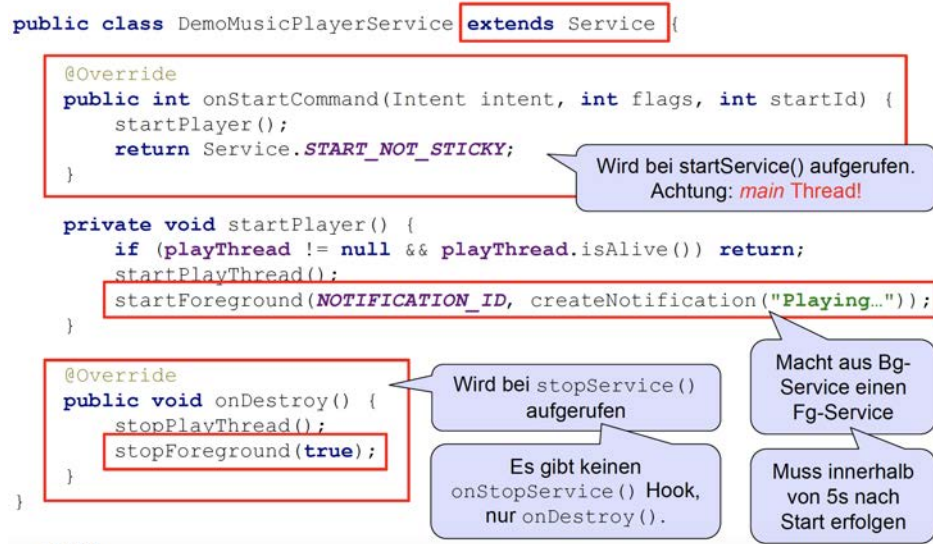


Abbildung 55: Demo eines Foreground Service - 01

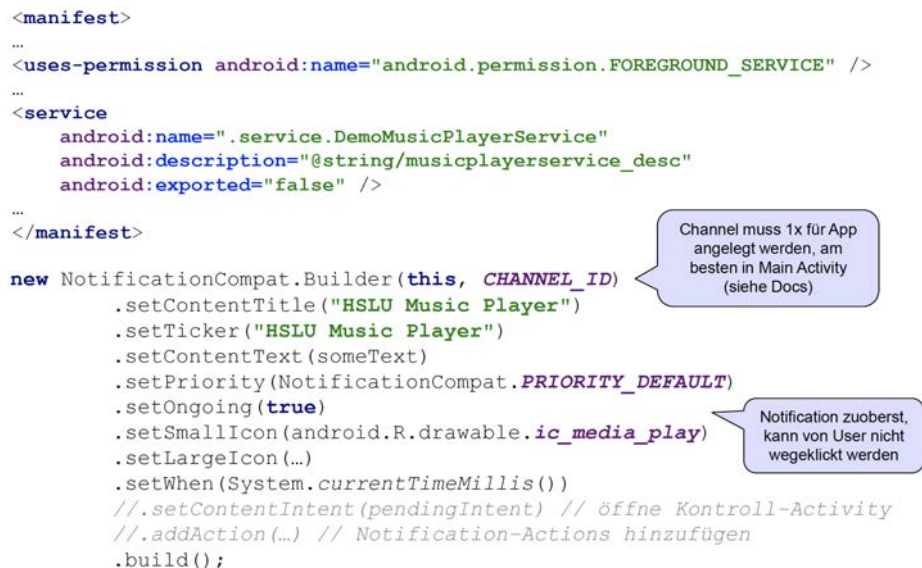


Abbildung 56: Demo eines Foreground Service - 02

Service starten:

(Im Intent können auch Parameter mitgegeben werden!)

```
1 public void startPlayerService(View v) {  
2     startService(new Intent(this, DemoMusicPlayerService.class));  
3 }
```

Service stoppen:

```
1 public void stopPlayerService(View v) {  
2     stopService(new Intent(this, DemoMusicPlayerService.class));  
3 }
```

5.1.4 Allgemeines Muster & „Stickyness“ von Services

- Ein Service wird beim App-Start oder beim Start eines Events als Foreground-Service gestartet und bleibt **alive**.
 - Der Service startet einen *Worker-Thread* oder *Thread-Pool*, der aktiv bleibt (zu Beginn ggf. idle).
 - Mittels `bindService()` kann synchron kommuniziert werden.
 - **Was soll mit einem Service passieren, wenn das System den App-Prozess zerstört und diesen später wiederherstellt?**
 - `onStartCommand()` retourniert die gewünschte Verhaltensweise, dies ist für gebundene Service jedoch nicht wichtig
- Mögliche Rückgabewerte von `onStartCommand()` wären:
- * **START_STICKY:**
Service nach Wiederherstellung automatisch wieder starten - `onStartCommand()` wird erneut aufgerufen, aber ohne Intent (Service sollte Zustand persistieren und wieder laden z.Bsp. Queue bei Musikplayer)
 - * **START_NOT_STICKY:**
Service nicht automatisch neu starten nach Wiederherstellung
 - * **START_REDELIVER_INTENT:**
Wie bei *START_STICKY*, der ursprüngliche Intent wird jedoch nochmals ausgeliefert, damit parametrisierte Reinitialisierung möglich ist.

5.1.5 Gebundene Services

- Service kann gebunden werden mittels `bindService(intent, connection, flag)`
- Client kommuniziert mit Service über `ServiceConnection`, damit kann die Funktionalität einer App exportiert werden (insbesondere mit einem „Remote Service“)
- Bindung lösen mit `unbindService(connection)`

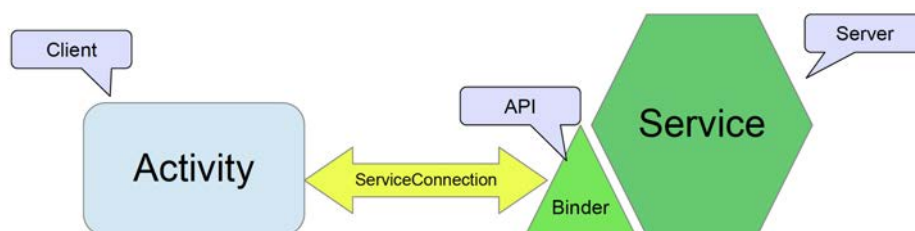


Abbildung 57: Verbindung bei einem gebundenen Service

Involvierte Klassen bei gebundenem Service (Die Namen der Klassen leiten sich von der dazugehörigen MOBPRO-Übung der SW05 ab und dienen bloss der Veranschaulichung der Klassen)

- **Service Interface** (MusicPlayerApi)
Definiert die API eines Service
- **Binder** (MusicPlayer.MusicPlayerApiBinder)
Implementiert das Service Interface und wird dem Client bei einer erfolgreichen Verbindung übergeben (Service-Stub / -Handle)
- **Service Connection** (MusicPlayerConnection)
Definiert die Callbacks für eine erfolgreiche/verlorene Verbindung und enthält Binder-Objekte (= API) bei einer erfolgreichen Verbindung
- **Service** (MusicPlayerService)
Implementiert `onBind(intent)` und gibt ein Binder-Objekt zurück
- **Client** (MainActivity)
Ruft `bindService(intent, connection(Callback-Handler), flag)`
(resp. `unbindService(connection)`) auf

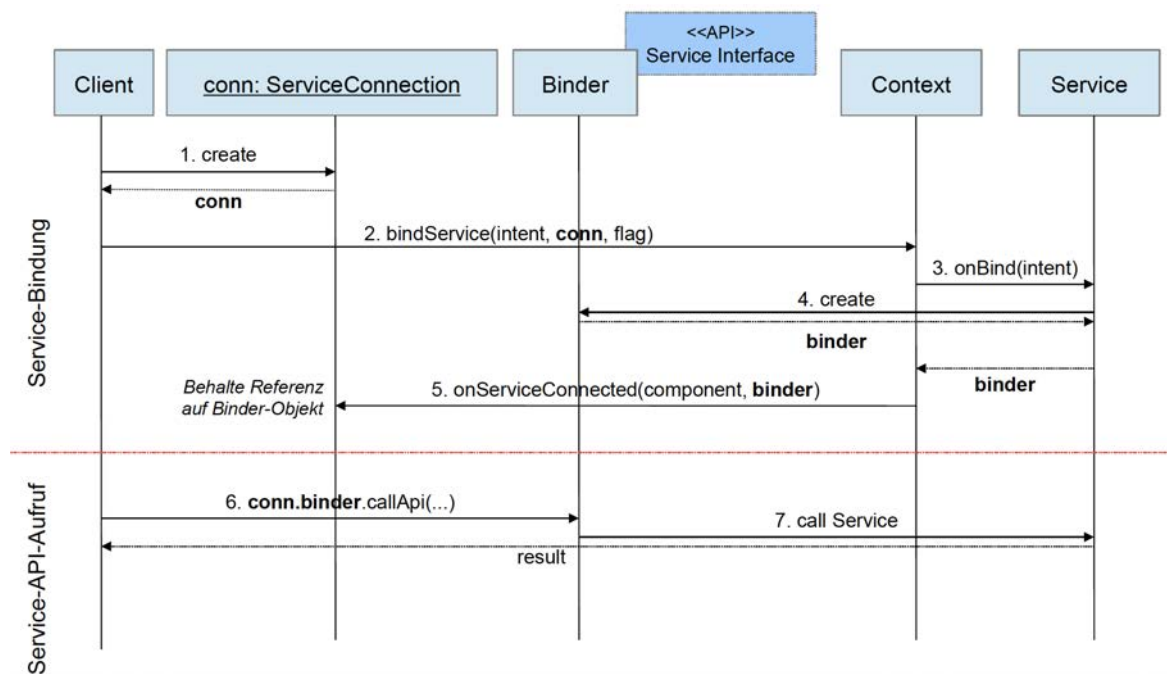


Abbildung 58: Ablauf Service-Bindung & Service API-Abruf

5.1.6 Fortführung Demo Service

Gebundener Service: Music Player Service (4)

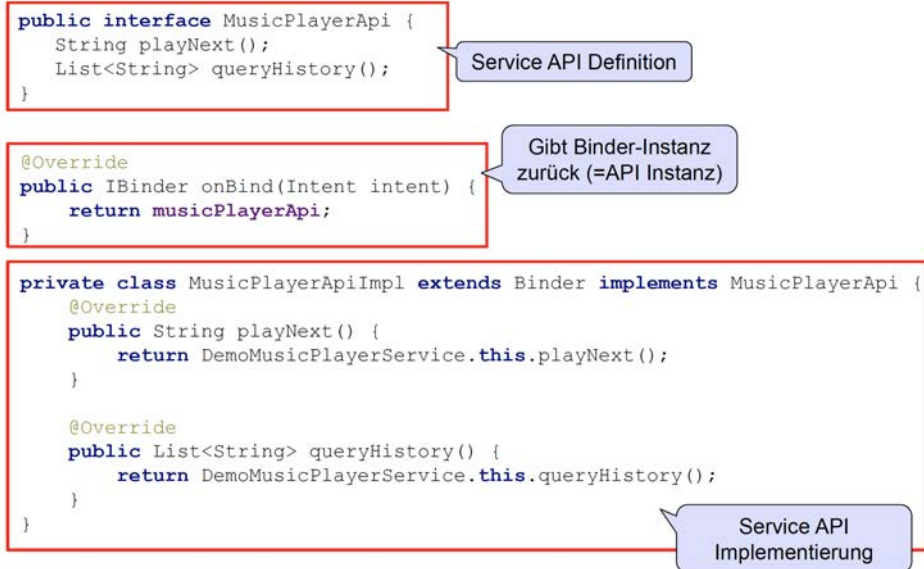


Abbildung 59: Demo eines Foreground Service - 04

Gebundener Service: Client (5)

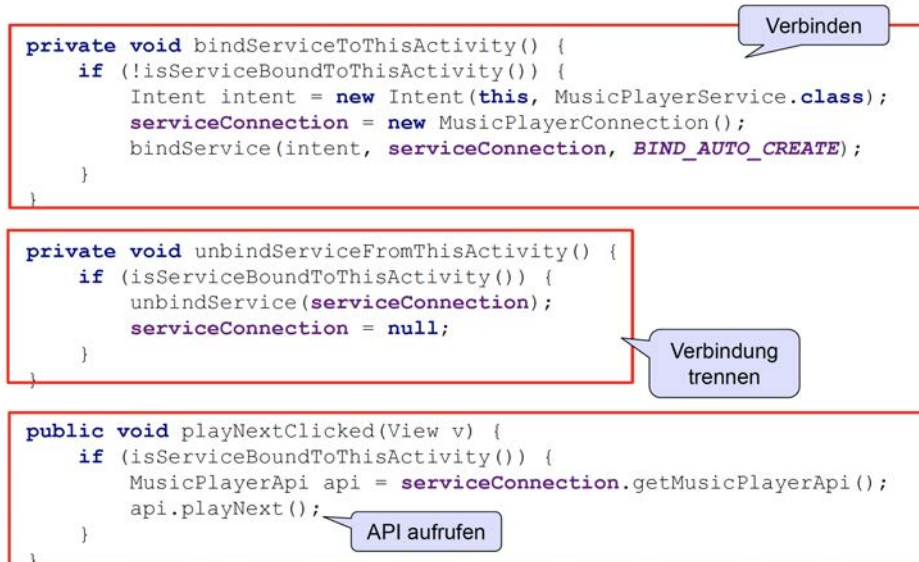


Abbildung 60: Demo eines Foreground Service - 05

5.2 Broadcast Receiver

- Broadcasts sind Nachrichten, es gibt einen app-internen „Message Bus“
- Alle Komponenten einer App können Broadcasts versenden und sich für den Empfang registrieren
- Das System selbst versendet ebenfalls Nachrichten bei gewissen Events (App installiert, Timer etc.)
- Aus Performancegründen stark eingeschränkt seit API 26, es werden nur noch sehr wenige Events global verteilt

5.2.1 Broadcasts versenden

- Broadcasts werden als Intents versendet
 - **Implizit** in der App via
`LocalBroadcastManager.getInstance(this).sendBroadcast(intent)`
 - **Explizit** an andere Apps über
`Context::sendBroadcast(intent)`
- Empfang von Broadcasts: Receiver (Empfänger)
 - Empfänger werden dynamisch im Code registriert:
`registerReceiver(receiver, filter)`
 - Können statisch im Manifest deklariert werden:
Tag `<receiver ...>` ← nur explizite Broadcasts, auch wenn App nicht gestartet

5.2.2 Globaler Broadcast Receiver

- Ein Broadcast Receiver ist immer nur so lange aktiv, wie die Bearbeitung der empfangenen Nachricht braucht.
(**DON'T**: keine AsyncTasks, keinen Service binden, keine Dialoge anzeigen!)
DO: Activity starten, Service starten, Notification senden)
 - Ansonsten wird er inaktiv und vom System gelöscht
 - Erneute Erzeugung nur auf Abruf
 - **Nur** für den Empfang von expliziten Broadcasts geeignet
- Broadcast Receiver hat kein UI:
Notifications für Kommunikation mit User benutzen oder einen Service starten für Hintergrundaufgaben

```
<receiver android:name=".BootCompletedReceiver">
  <intent-filter>
    <action android:name="android.intent.action.BOOT_COMPLETED" />
  </intent-filter>
</receiver>
```

Abbildung 61: Eintragen eines Receivers im Manifest

Dedizierter Broadcast Receiver Klasse erstellen:

```
1 public class BootCompletedReceiver extends BroadcastReceiver {
2     @Override
3     public void onReceive(Context context, Intent intent) {
4         // do something, when boot has completed, e.g. start a service...
5     }
6 }
```

Expliziten Broadcast an eine andere App versenden:

```
1 Intent broadcastIntent = new Intent("ACTION_MY_BROADCAST");
2 broadcastIntent.setPackage("ch.hs.lu.mobpro.other"); // Empfaenger ID
3 sendBroadcast(broadcastIntent)
```


5.2.3 Lokale Broadcasts - „App Message Bus“

Broadcast Receiver erzeugen & registrieren im Code:

```
1 downloadCompleteListener = new BroadcastReceiver() {
2     @Override
3     public void onReceive(Context context, Intent intent) {
4         Toast.makeText(this, "Got it!", LENGTH_SHORT).show();
5     }
6 };
7
8 IntentFilter filter = new IntentFilter("mobpro.DOWNLOAD_COMPLETE");
9 LocalBroadcastManager.getInstance(this)
10    .registerReceiver(downloadCompleteListener, filter);
```

Nachricht versenden (Emitter):

```
1 Intent downloadComplete = new Intent("mobpro.DOWNLOAD_COMPLETE");
2 downloadComplete.putExtra("file", "Terminator2.mp4");
   LocalBroadcastManager.getInstance(this).sendBroadcast(downloadComplete);
```

Broadcast Receiver deregistrieren (wenn er nicht mehr benötigt wird):

```
1 LocalBroadcastManager.getInstance(this)
2    .deregisterReceiver(downloadCompleteListener);
```

5.3 Work Manager & Broadcasts

- **WorkManager für Hintergrundtasks**

Repetitive oder einmalige Background-Tasks, die aufschiebbar sind, sollten via WorkManager erledigt werden.

- Wie findet die App heraus, wenn der Task abgeschlossen ist?

→ z.Bsp. mittels einem lokalen Broadcast (wird nur verarbeitet, wenn die App noch läuft bzw. solange der Receiver noch registriert ist, bspw. um eine Liste von heruntergeladenen Dateien zu aktualisieren)

- Beispielablauf mit WorkManager:

- Es ist ein lang andauernder Task vorhanden, bspw. Positionen von Objekten erkennen
- Dieser Worker-Task wird an den WorkManager übergeben
- Sobald die Positionen bestimmt wurden, werden diese mittels Broadcast zurückgemeldet
- Der Broadcast-Receiver hört auf diese Action und zeigt die Resultate in einem Toast

- Gradle Dependency:

```
androidx.work:work-runtime:2.0.0-rc01
```

- Nachfolgendes Beispiel funktioniert auch mit Threads

- Worker kann auch als wiederkehrender Task (auch mit anfänglichem Delay) registriert werden

- Arbeiten können auch in Graphen mit Abhängigkeiten definiert werden

The diagram illustrates the implementation of a WorkManager task. It consists of two main code blocks. The first block, enclosed in a red box, shows the `startBackgroundTaskClicked` method in an activity, which creates a `OneTimeWorkRequest` and enqueues it. A callout bubble points to this block with the text 'WorkRequest erstellen und erfassen'. The second block, also in a red box, shows the `LocalizeMissilesWorker` class, which extends `Worker`. It defines the `doWork` method, which logs, determines missile positions, creates an intent with the action `ACTION_LOCALIZE_MISSILES`, and sends a broadcast. A callout bubble points to the `doWork` method with the text 'Arbeit definieren'. Another callout bubble points to the broadcast sending line with the text 'Broadcast-Event mit Resultat verschicken'. A third callout bubble points to the `return` statement with the text 'Erfolg/Misserfolg melden'.

```
public void startBackgroundTaskClicked(View view) {
    OneTimeWorkRequest getLocationTask =
        new OneTimeWorkRequest.Builder(LocalizeMissilesWorker.class).build();
    WorkManager.getInstance().enqueue(getLocationTask);
}

public static class LocalizeMissilesWorker extends Worker {
    public LocalizeMissilesWorker(Context context, WorkerParameters params) {
        super(context, params);
    }

    @Override
    public Result doWork() {
        Log.i("LocalizeMissilesWorker", "Getting location of missile");
        // determine number and location of missiles (long time)

        Intent result = new Intent("mobpro.ACTION_LOCALIZE_MISSILES");
        result.putStringArrayListExtra("missilePositions", positions);
        LocalBroadcastManager.getInstance(getApplicationContext()).sendBroadcast(result);

        return Result.success(); // or Result.failure() or Result.retry()
    }
}
```

Abbildung 62: Beispielcode für einen WorkManager

6 Android 6 - Intents, Fragments, App-Widgets

6.1 Intent Filters

- Bei **impliziten Intents**:
Steht der Empfänger nicht im Vorherein fest, es gibt nur eine Nachricht mit einem „Anforderungsprofil“ für den Empfänger
- Das System eruiert mögliche Empfänger (Intent Resolution)
Drei mögliche Fälle:
 - Genau ein Empfänger gefunden → direkte Zustellung
 - Mehrere Empfänger → Auswahl durch User per Dialog
 - Kein Empfänger → `ActivityNotFoundException`
- Potentielle Intent-Empfänger deklarieren Intent-Filter im Manifest (oder deklarieren es in einem Broadcast Receiver)
- Das System vergleicht implizite Intents mit deklarierten Filtern und liefert die passenden Komponenten zurück
(Der Benutzer kann wählen, falls mehrere Filter passen und kein Favorit registriert ist)



Abbildung 63: Intent-Filter im Manifest deklarieren

Beispiel für eigene Intent-Action in den Vorlesungsfolien

6.1.1 Implizite Intents: Daten

Implizite Intents können (optional) folgende Daten enthalten:

- **Action**
Typ der Aktion, welche ausgeführt werden soll
Bsp. `ACTION_VIEW`, `ACTION_EDIT`, `CUSTOM_ACTION...`
- **Category**
Kategorie der Komponente, welche diesen Intent ausführen soll
Bsp. `DEFAULT`, `LAUNCHER`, `BROWSABLE...`
- **Data**
Beschreibung der Daten, mit welchen gearbeitet werden soll
(URI und Mime Type)
- **Extras**
Schlüssel / Wert-Paare für Zusatzinformationen

6.1.2 Implizite Intents: Auflösung von Intents

Das Android-System löst implizite Intents auf, indem die am besten zum Intent passende Komponente ausgewählt wird („Best Match“). Diese wird festgelegt durch Vergleich von:

- **Action:** Action des Intents muss im Filter sein
- **Category:** Jede Kategorie des Intents muss im Filter sein
- **Data (URI & Mime Type):** Alles im Intent unter „Data“ aufgelistete muss zum Filter passen

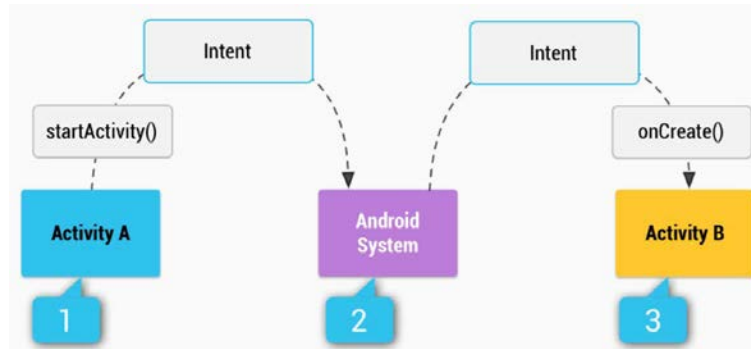


Figure 1. Illustration of how an implicit intent is delivered through the system to start another activity: **[1]** Activity A creates an **Intent** with an action description and passes it to **startActivity()**. **[2]** The Android System searches all apps for an intent filter that matches the intent. When a match is found, **[3]** the system starts the matching activity (Activity B) by invoking its **onCreate()** method and passing it the **Intent**.

Abbildung 64: Übertragen von Impliziten Intents zw. Applikationen (Google Doku)

Infos zu aktuell installierten Packages im Android-System können mit folgender Klasse abgefragt werden:

PackageManager

- `.query...()`: liefert alle passenden Intent-Auflösungen zurück
- `.resolve...()`: liefert die best-passenden Intent-Auflösungen zurück

Beispiel, um Activities für einen Intent abzufragen:

```
1 final List<ResolveInfo> resolveList = getPackageManager()
2     .queryIntentActivities(getHsluBrowserIntent(),
3         PackageManager.MATCH_DEFAULT_ONLY);
```

Aus dieser Liste für alle `ResolveInfo` das Feld `activityInfo.name` ausgeben, um alle verfügbaren Activities anzuzeigen.

6.2 Fragments

- **Fragments:**
Modularer (UI-)Teil innerhalb einer Activity
- **Eigenschaften:**
Eigener Lebenszyklus, eigener Zustand, kann einer laufenden Activity hinzugefügt / entfernt werden (Transaktion)
- Ist quasi eine Art „Sub-Activity“, welche in verschiedenen Activities wiederverwendet werden kann

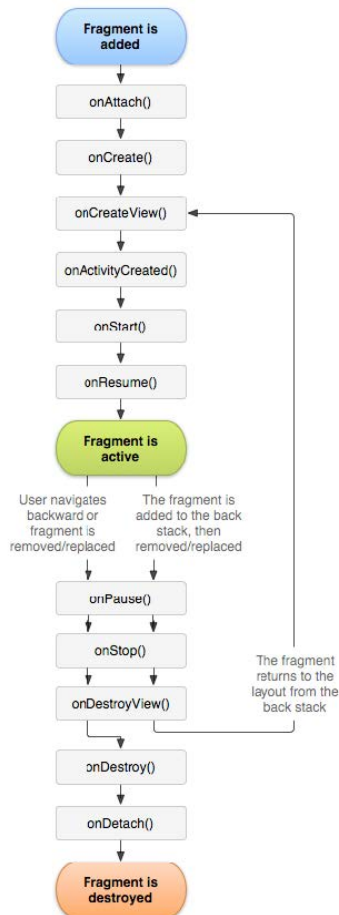


Abbildung 65: Lebenszyklus eines Fragments

- Fragments werden an Activities angehängt (attached)
- Fragments können im `layouts.xml` deklariert oder programmatisch erzeugt werden
- *FragmentManager* verwaltet die Fragments innerhalb einer Activity:
`Activity.getFragmentManager()`, mit Support Package:
`FragmentManager.getFragmentManager()`

6.2.1 Fragments hinzufügen

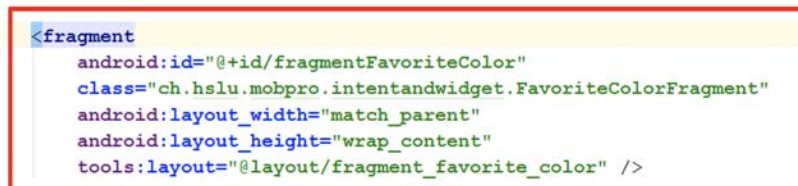
Mit dem `LayoutInflater` kann ein Fragment-UI aus einem `layout.xml` erzeugt werden. Die von Fragment abgeleitete Klasse implementiert die zugehörige Logik.

```
1 public static class ExampleFragment extends Fragment {  
2     @Override  
3     public View onCreateView(LayoutInflater inflater, ViewGroup container,  
4         Bundle savedInstanceState) {  
5         return inflater.inflate(R.layout.example_fragment, container, false);  
6     }  
7 }
```

In einer Activity können Fragments programmatisch (ggf. im `onCreate()`) hinzugefügt werden. Dies passiert mittels `FragmentTransactions`, welche vom `FragmentManager` verwaltet werden. (*fragmentContainer ist in diesem Beispiel ein `FrameLayout` im `layout.xml` der Activity*)

```
1 getFragmentManager()  
2     .beginTransaction()  
3     .add(R.id.fragmentContainer, FavColorFragment.newInstance())  
4     .commit();
```

Fragments können auch direkt im `layout.xml` einer Activity deklariert werden:



```
<fragment  
    android:id="@+id/fragmentFavoriteColor"  
    class="ch.hslu.mobpro.intentandwidget.FavoriteColorFragment"  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    tools:layout="@layout/fragment_favorite_color" />
```

Abbildung 66: Deklaration eines Fragments im `layout.xml`

Zum Vorlesungsbeispiel: Einstellungen des Fragments werden in `SharedPreferences` gespeichert, wobei die Prefs in `onResume()` wieder geladen werden, damit diese sich bei einem Wechsel synchronisieren.

6.3 App-Widgets

- App-Widgets sind quasi „Mini-Apps“ auf dem Homescreen eines Androidgeräts, zeigen dort wichtige Daten und Funktionalitäten einer App an
- Es gibt verschiedene Typen: Information / Collection / Control / Hybrid-Widgets
- **How-To für Widgets**
 - Widget im Android-Manifest als solches deklarieren
→ Widgets sind spezielle `BroadcastReceiver`, werden als Receiver-Komponenten deklariert
 - `AppWidgetProviderInfo` muss angegeben werden:
 - * *Deklarativ:* `widget_provider_info.xml`
 - * *Programmatisch:* Klasse `AppWidgetProviderInfo`
 - UI für das Widget in einer `widget.xml` Datei
 - Eigene von `AppWidgetProvider` abgeleitete Klasse, behandelt Aktualisierungslogik etc.
- **Codebeispiele und wie ein Force Update für ein Widget programmiert wird, ist auf den Folien zu „MobPro-Android_6“ auf Seiten 29-34 ersichtlich.**

6.4 App-Design

Unter *Design für Android* und *Material Design* können die ständig aktualisierten Design-Vorgaben eingesehen werden. Für wichtigste Interaktionen verwenden Apps auf Android oft die App-Bar. Das Android Asset Studio bietet zudem eine Auswahl an Tools, um einfach Assets wie Launcher- / Notification- / App-Bar-Icons und vieles mehr zu erstellen.

6.5 Usability & Prototyping

- Prototypen helfen beim Design (iterativer Prozess)
- Kommunikation mit Kunden, der Benutzer sieht etwas
Verständnis für EntwicklerIn für die App/Anwendungsdomäne
- Stufen im Designmodell:
(→ Analyze ↔ Design ↔ Prototype → Evaluate →) → Final Product
- Prototypen in Software oder auf Papier, durcharbeiten für besseren Überblick der App-Funktionalität und -Navigation (Wireframes / Mocks / Storyboard ...)
- Android-Doku zu Wireframes zur Vertiefung anschauen!

6.6 Android Jetpack & Support Library (AppCompat)

„When developing apps that support multiple API versions, you may want a standard way to **provide newer features on earlier versions of Android** or gracefully fall back to equivalent functionality. Rather than building code to handle earlier versions of the platform, you can leverage these libraries to provide that compatibility layer. In addition, the Support Libraries provide additional convenience classes and features not available in the standard Framework API for easier development and support across more devices. Originally a single binary library for apps, the Android Support Library has evolved into a suite of libraries for app development. Many of these libraries are now a strongly recommended, if not essential, part of app development.“

Beispiel: Action Bar (App Bar) Die Action Bar gibt es erst seit API 11, ist dank der Support Library aber schon ab API-Level 7 (Android 2.1) verfügbar (Warnhinweis dazu in der Support Library Doku).

Android Jetpack Ist verfügbar seit API 28, enthält die AppCompat-Library, man muss jedoch seine App zu AndroidX migrieren (wird wohl erst ab Android 10 standardmässig integriert sein), Infos dazu auf entsprechender Android-Doku Webseite.

Beispiele für Hilfslibraries: *Databinding, LiveData, ViewModel, Room, Download Manager, Notifications, Sharing, Animation & Transitions, Layout, ...*

6.7 Publizieren von Android-Apps

Primäre Quellen für App-Veröffentlichung unter *Nützliche Links*

Jeder kann Apps im Google Play Store veröffentlichen, Voraussetzungen dafür sind:

- gratis Google Account
- Google Play Publisher Account (25\$, einmalig), neuen Google Acc dafür erstellen
- Google Payments Merchant Account (nur für kostenpflichtige Apps, 30% an Google, 70% an EntwicklerIn)

Grob-Vorgehen bei der Veröffentlichung:

1. **Code Cleanup, Release vorbereiten**
Verionierung, Internationalisierung, Logging/Debugging entfernen etc.
2. **Release erstellen & signieren** (Dev. Identity o.ä.)
3. **Werbematerial vorbereiten**
Screenshots, Hi-Res-Icons, Promo, Website, Video, Google Ads etc.
4. **Distributionseinstellungen setzen**
Content Rating, Verfügbarkeit, Grösse, Bezahlungsart (gratis, paid, in-app)
5. **Release Upload & Veröffentlichung**
6. Link: *Offizielle Google Launch Checklist*

Veröffentlicht wird eine APK-Datei (kurz: signiertes Archiv mit einer App drin)

ProGuard ist ein Optimierer, welchen Code schwer lesbar und kleiner macht und ist per Default im Release-Modus aktiviert. Wird in Zukunft wahrscheinlich durch Google's eigenen R8 Optimierer ersetzt.

6.7.1 Signieren von Apps

- Play-Store Apps müssen signiert sein
(Nicht mit (default) Developer Key aus der IDE)
- Self-Signing ist erlaubt (kein autorisiertes Zertifikat notwendig)
- Benötigt wird ein Keystore mit Private Keys
- Android Studio Wizard hilft beim Erstellen eines Keystores mit privatem Schlüssel (= Zertifikat)
- Updates müssen mit dem **gleichen** Zertifikat signiert sein (produktiven Schlüssel nicht verlieren!)
 - Wenn ja: problemloses Update möglich
 - Wenn nein: dann muss App mit neuem Package-Namen als komplett neue App installiert werden

6.7.2 Andere Distributionsmöglichkeiten

Neben Google Play Store können Apps in anderen Stores oder über eine eigene Website oder andere Quelle verteilt werden. Das Gerät, welches die APK installieren möchte, muss dabei aber die Installation aus unbekannten Quellen in den Einstellungen erlauben.

6.7.3 Test before Release

Apps manuell und automatisch testen. Manuell auf mindestens einem Gerät, im besten Falle auf mehreren Geräten mit unterschiedlichen Bildschirmauflösungen, Android-Versionen und verschiedenen Herstellern. Grosser Aufwand, dass eine App auf möglichst vielen Geräten lauffähig ist!

7 Android 7 - Hybrid WebApp

7.1 Mobile Web-Anwendungen

- **Web-App**

JavaScript, HTML, CSS mit Responsive Design, wird im Browser ausgeführt

- **Hybrid**

Web-App in nativem Wrapper verpackt, Konnektor-Plugins für die Nutzung nativer Services, kann als Native App installiert werden (Bsp. Cordova / PhoneGap)

7.1.1 Charakteristiken für eine Web-App

- Eine Web-Anwendung für mehrere/alle Plattformen
- Voraussetzung pro Plattform ist nur ein Web-Browser
- Verwenden meist JavaScript-Web-Frameworks
Bsp. Ionic, jQuery Mobile, Titanium etc.

Vorteile einer Web-App:

- Eine Codebasis für mehrere/alle Plattformen
- Kein App-Store, Download oder Installation notwendig
- App kann jederzeit veröffentlicht und verändert werden
- Eine vorhandene Web-App kann zu einer mobilen App erweitert werden (Responsive / Progressive)

Nachteile einer Web-App:

- Läuft nur im Browser („Rahmenapplikation“)
- Kein Zugriff auf alle Funktionen (Kontakte, externe Hardware etc.)
(HTML5 ermöglicht immer mehr solcher Zugriffe (Storage, Kamera, GPS etc.))
- App nicht im Store, Benutzer müssen diese erst finden (Marketing benötigt)
(Kein installiertes App-Icon, ggf. Browser-Lesezeichen auf Homescreen)

7.1.2 Charakteristiken für eine Hybride App

- Web-Anwendung für mehrere/alle Plattformen, verpackt in einem nativen Container, welcher in einer WebView läuft und durch einen nativen Controller installierbar wird
- Zugriff auf native Features durch Plugins, können auch selber geschrieben werden
- Pro Plattform wird ein nativer Container vorausgesetzt
Bsp. Apache Cordova (Open Source), PhoneGap (kommerziell), Appcelerator Titanium

Vorteile einer Hybrid App:

- Grundsätzlich eine Codebasis für alle Plattformen
- App in einem Store auffindbar, gewisse App-Qualität & -Sicherheit
- Zugriff auf viele Gerätefunktionen dank nativer APIs
- Native Look & Feel (je nach Technologie)

Nachteile einer Hybrid App:

- Kein garantierter Zugriff auf alle Gerätefunktionen
- Unterstützung verschiedener Plattformen kann aufwendig sein, ggf. sind spezifische Anpassungen / Code nötig (evtl. selber Plugins schreiben)

7.1.3 Warum eine App als Web-App entwickeln?

- Läuft grundsätzlich auf allen Plattformen
- Für alle Applikationen derselbe Update-Prozess (einfacher)
- Developers: Web- vs. Native Know-How
- Kosten sparen, da App nur einmal entwickelt werden muss (nicht einmal pro Plattform)
- Leichtgewichtig, da keine Installation nötig ist (evtl. bei One-Time-Usage)
- Synergien: ggf. existierende Desktop-Web-Plattform
- Immer mehr native Fähigkeiten mit HTML5 (Kamera, Persistenz, Location etc.)

7.1.4 Probleme von Webseiten auf Mobil-Geräten

- Limitierte Display-Größen
- Portrait / Landscape Modus
- Interaktion: Touch \neq Maus, Gesten, Gyro etc.
- Unterschiedliche Browser (CSS, JavaScript)
- Kein Zugriff auf Hardware-Ressourcen (Kamera, GPS, Sensoren etc.) und Software-Ressourcen (Kontakte etc.)

Eine Reihe an **Toolkits** adressieren diese Probleme und helfen bei deren Behebung:
Bootstrap, Sencha Touch, jQuery (Mobile), **ionic**, **Cordova** / **PhoneGap**, u.v.m.

7.1.5 Info zu Grundlagen HTML, CSS, JavaScript

Grundlagen für HTML, CSS und JavaScript sind in MOBPRO Folien ersichtlich in **MobPro_WebHybridApp** Seiten ca. 18 - 29 (oder mach dich einfach im Internet kurz schlau darüber)

7.2 Mobile App-Entwicklung mit Ionic

Ionic bietet UI-Komponenten, APIs etc. inklusive Projektvorlagen (App-Layouts & Navigation). Ionic benutzt unter anderem Angular, TypeScript & Cordova. Eine Übersicht über alle CLI Commands für Installation & Ausführung ist unter *Nützliche Links* vermerkt.

7.2.1 Ionic: Projektstruktur

App-Seiten sind Angular-Module in TypeScript (.ts) geschrieben. Diese liegen im Projekt in `src/app/pages`. Müssen aber **registriert** werden unter `src/app/app.module.ts`!

Angular Ein von Google entwickeltes Web-Framework mit Dependency Injection für Komponenten-basierte Entwicklung von Web-Apps.

Achtung: AngularJS \neq Angular! (erste, inkompatible Version)

TypeScript Ein von Microsoft entwickelter JavaScript-Dialekt mit statischem Typing, Klassen und Interfaces. Wird nach JavaScript kompiliert, sieht also im Browser wieder anders aus.

7.2.2 Ionic: Seitenaufbau

Kurz: eine Seite einer Ionic-App besteht aus

- `*.html`: Seitenaufbau mit Struktur
Besteht aus: ion-header, ion-content, ion-footer
- `*.scss`: Erscheinung / Design der Seite
- `*.ts`: App-Logik für eine Seite

7.2.3 Ionic: How-To's

Hinweis: Ich schreibe hier nur die nützlichen Infos aus den Folien auf, für Code-Beispiele wieder Folien aus der Vorlesung konsultieren (Seiten 48 - 56)

http-get `httpClient.get(http://...)` ist asynchron, liefert als Resultat ein `Observable` zurück. Mit `.subscribe(data =>)` dieses Resultats wird ein Callback (`data =>`) gesetzt, welcher ausgeführt wird, wenn die Daten da bzw. vorhanden sind.

Die Funktionalität im Beispiel ist direkt im `page.ts` eingebaut, besser wäre diese in einen Service (Provider) auszulagern. Dazu muss das Modul `HttpClientModule` im File `src/app/app.module.ts` unter `imports` eingetragen werden.

parse JSON in TypeScript Die Grundidee hierbei ist, die erhaltenen JSON-Daten auf ein passendes TypeScript-Interface zu mappen (ähnlich wie bei Retrofit und Gson). In diesem Interface nur die gewünschten JSON-Properties aufnehmen, die anderen können einfach weggelassen werden.

do data binding Im `.html` kann bspw. ein Wert `title` eingegeben werden, welcher im `.ts` deklariert wird und einen Wert zugeteilt bekommt. Dies gehört zur Basis-Funktionalität von Angular.

use HTML input field Für Text-Input kann im `.html` ein `ion-input` eingefügt werden.

Mithilfe von bspw. `[(ngModel)] = „searchQuery“`, welches im `.ts` abgerufen wird, können eingegebene Daten direkt zur Verarbeitung übergeben werden.

push another page Auf `(click)` eines Buttons im `.html` kann eine Funktion aus `.ts` gebunden werden, welche mit `.push(page)` dann eine andere Seite „pusht“ bzw. aufruft.

use dynamic lists (aka *ngFor) Mit `*ngFor` im `.html` kann über die List-Property eines Controllers iteriert werden, um so wie im Beispiel eine Liste von Filmen auf einer Page anzuzeigen.

call method on controller on event Einer Funktion im `.ts` kann ein Parameter übergeben werden, z.Bsp. um die DetailPage eines ausgewählten Filmes zu pushen. Für genauere Infos schaut in die Folien oder Ionic Doku.

7.3 Hybride Web-Apps mit Cordova (Native Wrapper)

Cordova basiert auf HTML5, CSS3, JavaScript und nativem Code. Unterstützt mittels Plugins den Zugriff auf native APIs (bspw. für Zugriff auf Ressourcen des Geräts)

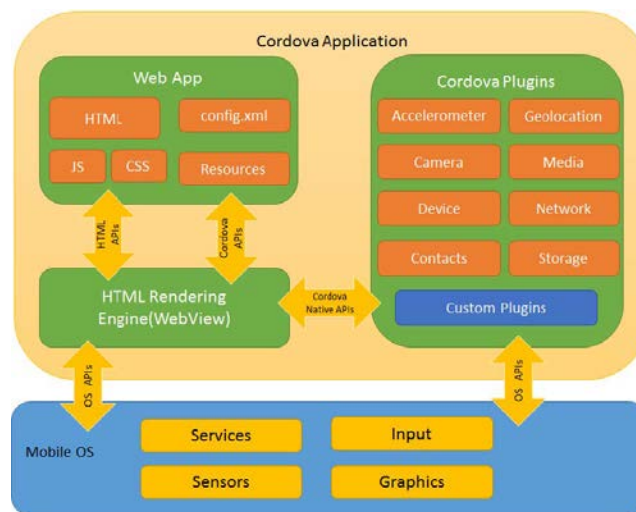


Abbildung 67: Cordova-Architektur im Überblick

7.3.1 Cordova vs. Phonegap

Cordova hiess früher Phonegap, bis die Firma dahinter von Adobe aufgekauft wurde. Die Codebase wurde Apache überlassen, welche das Projekt unter dem Namen Cordova weiterführten. Phonegap ist nun eine Distribution von Cordova, welche eine einfachere Einbindung in Adobe Services erlaubt. Beide Distributionen, Phonegap und Cordova, sind freie Software (Apache Lizenz 2.0)

7.3.2 Cordova & Ionic

Cordova ist in Ionic integriert und kann direkt zusammen installiert werden. Mit Plugins ist es möglich, dank Cordova auf native Möglichkeiten (APIs, HW) der Plattform zuzugreifen. Die Liste mit allen Plugins ist unter *Nützliche Links* zu finden.

cordova build	Build (prepare + compile) an Ionic project for a given platform
cordova compile	Compile native platform code
cordova emulate	Emulate an Ionic project on a simulator or emulator
cordova platform	Manage Cordova platform targets
cordova plugin	Manage Cordova plugins
cordova prepare	Copies assets to Cordova platforms, preparing them for native builds
cordova requirements	Checks and print out all the requirements for platforms
cordova resources	Automatically create icon and splash screen resources
cordova run	Run an Ionic project on a connected device

Abbildung 68: Alle verfügbaren CLI-Commands für Cordova

8 Android 8 - Entwicklungsansätze

8.1 Anforderungen für Entwicklung mobiler Apps

- App-Entwickler und Designer müssen ihr „Mindset“ an alle neuen Möglichkeiten anpassen. (*Touch, Drag, Pinch, Location, Orientation, NFC etc.*)
- Anforderungen und Wünsche der Nutzer & des Markts prüfen! (*Benutzerinteraktion, Plattform-Standards etc.*)
- Plattformen (Geräte inkl. Sensoren) und Betriebssysteme (APIs) müssen den Entwicklern bekannt sein

8.2 Entwicklung mobiler Apps

- **Web-App**
JavaScript, HTML, CSS mit Responsive Design, wird im Browser ausgeführt
- **Hybrid**
Webapp in nativem Wrapper „verpackt“, mit Konnektor-Plugins, kann als native App auf einer Plattform installiert werden (Bsp. Cordova, Phonegap)
- **Cross-compiled**
In Sprache X geschriebene App, die nach Java oder Objective-C (oder direkt binäres Format) kompiliert und somit „native“ wird (Bsp. Xamarin: C# / Ruby)
- **JIT-compiled / VM**
In JavaScript geschriebene App, die auf der JS-Engine des Zielsystems läuft und dort just-in-time (JIT) kompiliert wird. Verwendet natives GUI und Konnektoren (Plugins), um mit der nativen Plattform zu interagieren (Bsp. NativeScript)
(Daneben gibt es noch Game Engines bspw. Unity, welche ebenfalls JIT-compiled sind, laufen auf einer sehr spezifischen VM)
- **„Echte“ Native App**
Spezifisch programmiert für ein mobiles OS (iOS, Android). Kann das volle Featureset der Zielplattform ausnutzen (inkl. der neuesten Features)

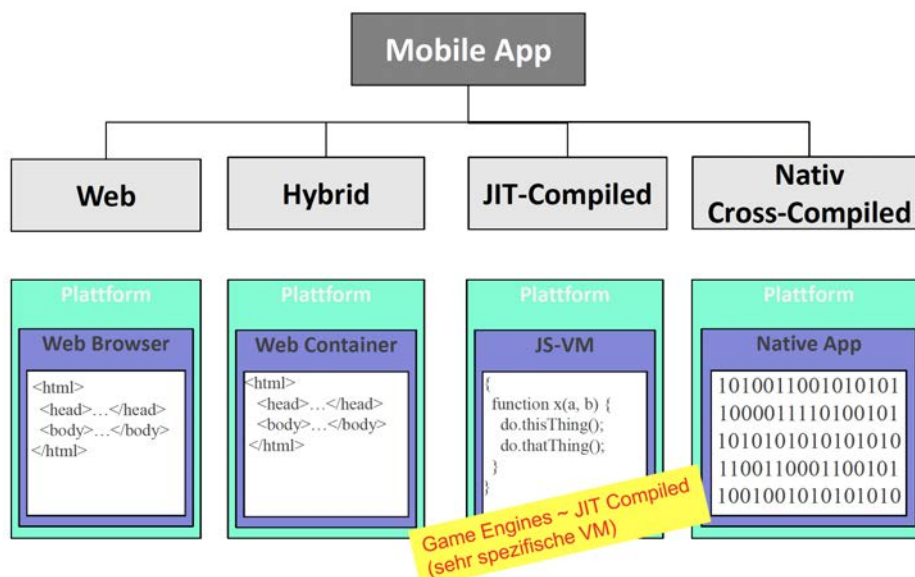


Abbildung 69: Entwicklungsansätze für Mobile Apps

8.3 Charakteristiken für eine Native App

- Die Applikation wird in der von der Plattform vorgegebenen Sprache (inkl. vorgesehene APIs) entwickelt (pro mobile Plattform wird eine eigenständige Applikation entwickelt)
- Voraussetzungen pro Plattform: Natives API + SDK
Single-Plattform-Ansatz, volle Ausnutzung plattformspezifischer Funktionen, Natives Look & Feel inkl. Native UI Patterns
- *Apple*: Objective-C / Swift
Android: Java
Windows: C#

Vorteile einer Native App:

- Zugriff auf alle nativen Möglichkeiten
- Natives Look and Feel
- Typischerweise die schnellste Lösung für Appentwicklung
- App-Stores: App auffindbar, gewisse App-Qualität & -Sicherheit

Nachteile einer Native App:

- Grosser Entwicklungsaufwand, da für eine Plattform je eine App entwickelt werden muss, mehrere Plattformen zu unterstützen ist teuer, die gleiche Funktionalität muss evtl. doppelt oder mehr entwickelt werden, kein Code-Sharing
- Benutzer können verschiedene OS-Versionen haben, ist aufwändiger zu unterstützen
- App-Store-Zulassung kann die Veröffentlichung verlangsamen oder verhindern (verglichen mit reinen Web-Apps)

8.4 Charakteristiken für Cross-/JIT-Compile Apps

- Cross-Plattform-Ansatz: Single-Code-Base für alle Plattformen mittels Cross-Compile
- Keine/wenig Kenntnis der nativen Plattform notwendig, Abstraktion, eigenes App-Modell, grösstes gemeinsames Vielfaches an Funktionen
- Installierbar und in einem Store
- **Cross-Compile**: Wird nach der Programmierung in nativen Code oder Binary umgewandelt
- **JIT-Compile**: Läuft auf JavaScript-VM auf der Zielpattform, just-in-time Kompilierung
(*JS-VM ist heutzutage auf jeder Plattform vorhanden und i.d.R. up-to-date/schnell/modern via Platform Updates*)

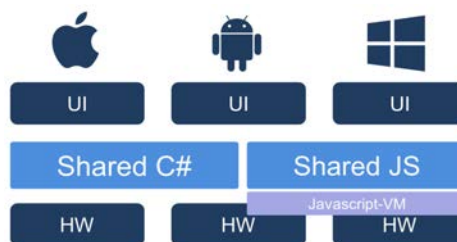


Abbildung 70: Silo-System bei Cross-/JIT-Compile Apps

Flutter Für Cross-Platform-Development, entwickelt von Google (v1.0 Release am 4. Dezember 2018). Eine Code-Base für alle Plattformen (iOS & Android). Läuft mit eigener Sprache (Dart, von Google entwickelt), hat zwar kein natives UI, zeichnet das UI jedoch selber (ist somit ähnlicher zur Web-App), es wird trotzdem „native performance“ versprochen. So werden eigene Widgets möglich (Material-Look auf Android, Native Look auf iOS wird imitiert). Läuft JIT-compiled auf eigener VM (Flutter Engine)

Vorteile von Cross-/JIT-Compile Apps:

- Single Code-Base (Funktionalität muss nur einmal implementiert werden)
- Vorhandenes Entwicklerknowhow (Web, C# etc.) wiederverwendbar, keine Einstiegsbarrieren (ausser evtl. bei Dart)
- Geringere Entwicklungskosten
- Sieht aus wie Native und läuft gleich schnell (kein Geflicker)
- *Eigenes Applikationsmodell \neq Native Modell*

Nachteile von Cross-/JIT-Compile Apps:

- *Eigenes Applikationsmodell \neq Native Modell*
- Eingeschränktes Programmiermodell nach dem „grössten gemeinsamen Vielfachen“
- Native Features sind uneingeschränkt nutzbar, aber aufwändig zu nutzen
- Ggf, schwieriger zu Debuggen

8.5 Native Android-Programmierung

- Programmiersprache: Java
- App läuft in einer VM:
Android Runtime (ART) für Android \geq 5.0
Dalvik VM für Android $<$ 5.0
- IDE: Android Studio, IntelliJ, (Eclipse + ADT) etc.
- Entwicklerkosten für Store-Publikation: 25\$ pro Entwickler (einmalig)
- App-Vertrieb (Google Play, andere Stores, Websites etc.): 3.4 Mio.+

8.6 Native iOS-Programmierung

- Programmiersprache: Objective-C, Swift (C, C++)
- App läuft nativ auf iOS (binary Code \neq VM)
- IDE: Xcode, (AppCode)
- Entwicklerkosten für Store-Publikation: 99\$ pro Entwickler (jährlich)
- App-Vertrieb (App Store, Enterprise Deployment): 2.2 Mio. +

Eigenschaft	Web-App	Hybride Web-App	Cross-Platform	Native App
Plattformabhängig	Nein (aber: Browser abhängig)	Nein (Grundsätzlich)	Nein (Grundsätzlich)	Ja
Entwicklungsaufwand	1 App total	1 App + Anpassungen pro Plattform	1 App + Anpassungen pro Plattform	1 App pro Plattform
Nutzung aller nativen Möglichkeiten (OS & HW)	Nein (eher weniger als bei hybrid)	Nein (eher mehr als bei Web-App)	Jein (grundsätzlich alles möglich, aber nicht out-of-the-Box)	Ja
Vermarktung	Bannerwerbung, Abo	App-Store, Bannerwerbung, Abo	App-Store, Bannerwerbung, Abo	App-Store, Bannerwerbung, Abo
Vertrieb	Mobiler Browser	App-Store	App-Store	App-Store
App im Store	Nein (Aber App-Icon auf Homescreen möglich)	Ja	Ja	Ja
Performanz	Eher schlecht	Je nach Ansatz, eher schlechter als nativ	Gut	Gut

Abbildung 71: Übersicht über die mobilen Entwicklungsansätze

8.7 Single vs. Cross Plattform Entwicklung

Produkt	Sprache(n)	Ansatz
Android nativ	Java, Kotlin	nativ
Blackberry nativ	Java	nativ
iOS nativ	Objective-C, Swift	nativ
WindowsPhone nativ	C#	nativ

Abbildung 72: Native Produktbeispiele für Single-Plattform Entwicklung

Single-Plattform-Entwicklung folgt immer dem nativen Ansatz (man entwickelt pro Plattform spezifisch eine App in der vorgegebenen Sprache).

Produkt	Sprache(n)	Ansatz
Fire Monkey	Delphi	nativ
PhoneGap / Cordova	JS, HTML, CSS	hybrid
Qt	C++, QML (JS)	nativ: C++
React Native	JS	hybrid / nativ
Titanium	JS, XML, CSS	hybrid
Website „GWT“	Java, HTML, CSS	Web
Website „Ionic“	JS, CSS, HTML	Web
Flutter	Dart	jit-compiled
Xamarin	C#	nativ

Abbildung 73: Produktbeispiele für Cross-Plattform Entwicklung

Cross-Plattform-Entwicklung erzeugt je nach Produkt hybride, native, jit-compiled oder Web Apps.

8.8 Szenarien & Fazit

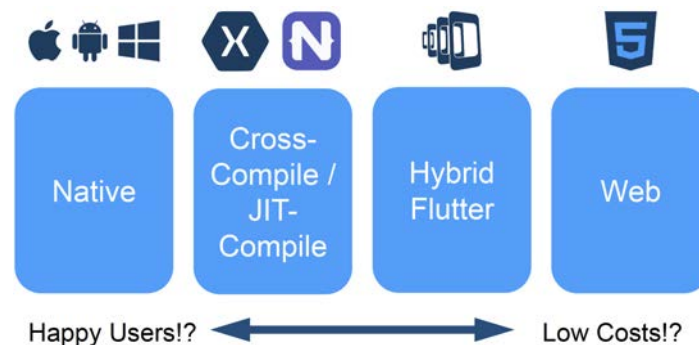


Abbildung 74: Das Spannungsfeld in der Mobilen Entwicklung

Cross-, JIT-compiled und hybrid werden typischerweise für Cross-Compile-Entwicklung eingesetzt. Markante Unterschiede dabei sind:


- **Cross-Compile:** erzeugt eine native App
- **Hybrid:** eine Web-App in einem nativen Container
- **JIT-Compile:** liegt irgendwo dazwischen

Kurz: für mobile Apps gibt es 3 Ansätze: Nativ, Hybrid oder Web.

8.9 Bester Ansatz?

Die bestmögliche Lösung ist abhängig von den Anforderungen & Möglichkeiten. So wird stets nach Szenario unterschieden. Single-Plattform bzw. nativ eignet sich am besten, wenn genügend Zeit und Geld vorhanden sind. Cross-Plattform hat grundsätzlich Limitierungen und „hinkt“ den OS hinterher. Beispiele:

- Info-App ohne viel Interaktionen
→ **Web-App**
- Komplett natives Look & Feel
→ **Nativ (SP & CP)**
- Viele Plattformen nativ unterstützen, preiswert, .NET- oder Angular-Knowhow vorhanden
→ **Cross- / JIT-Compile (Xamarin, NativeScript)**
- Gemeinsame Code-Basis, Cross-Plattform, eigene Widgets, Hot Reload, Kleine App / Prototyping
→ **Flutter**
- HW-Features werden auf verschiedenen Plattformen benötigt (NFC, Kamera, Storage etc.)
→ **Hybrid / Nativ**



Eigenschaft	Native SP	Cross-Plattform	Hybrid Web	Web
Performance	Gut	Mittelmässig	Schlecht	Schlecht
Look & Feel ⁽¹⁾	Gut	Mittelmässig	Schlecht	Schlecht
Zugriff auf Gerätefunktionen ⁽²⁾	Gut	Mittelmässig	Schlecht	Schlecht
Portabilität von Code	Schlecht	Mittelmässig	Gut	Gut
Anzahl benötigter Technologien ⁽⁴⁾	Schlecht	Mittelmässig	Gut	Gut
Wiederverwendung von Code ⁽⁴⁾	Schlecht	Mittelmässig	Gut	Gut
Upgrades ohne Einschränkungen ⁽³⁾	Schlecht	Schlecht	Schlecht	Gut
Installationserlebnis ⁽³⁾	Gut	Gut	Gut	Schlecht
Offline Nutzung	Gut	Gut	Gut	Mittelmässig
Veröffentlichung (Gebühr, Regeln)	Mittelmässig	Mittelmässig	Mittelmässig	Gut

Gut

Mittelmässig

Schlecht

(1) CP, Hybrid & Web Apps brauchen für natives Look & Feel Bibliotheken

(2) Hybride Apps greifen via native Bibliotheken auf Gerätefunktionen zu

(3) Native / Hybrid werden via App Store vertrieben (Prüfungsverfahren)

(4) Im Hinblick auf verschiedene Plattformen, für JIT immer gut

Abbildung 75: Vergleich der verschiedenen Ansätze

Alles hat Vor- und Nachteile. Web- und hybride Ansätze eignen sich oft, haben aber auch Nachteile gegenüber SP-Entwicklung. Cross-Compile kann bspw. Sinn machen für „.NET-Firmen“. Solange ein Benutzer ein Gerät (bzw. das OS) unterscheiden kann, wird es immer ein Verlangen nach native Apps geben.

9 Nützliche Links

9.1 Android 1 - Grundlagen

- **Referenz-Liste aller Android-Versionen**
https://de.wikipedia.org/wiki/Liste_von_Android-Versionen
- **Android Developers | Startseite**
<https://developer.android.com/>
- **Android Developers | Developer Guide**
<https://developer.android.com/guide>
- **Android Developers | Package Index (APIs)**
<https://developer.android.com/reference>
- **Android Developers | Android Platform Architecture**
<https://developer.android.com/guide/platform/>

9.2 Android 2 - Benutzerschnittstellen

- **Android Developers | ViewGroup.LayoutParams**
<https://developer.android.com/reference/android/view/ViewGroup.LayoutParams.html>
- **Android Developers | Unterschiedliche Screen Sizes**
<https://developer.android.com/training/multiscreen/screensizes.html>
- **Android Developers | Debug mit Layout Inspector**
<https://developer.android.com/studio/debug/layout-inspector.html>
- **Responsive UI mit ConstraintLayout**
<https://developer.android.com/training/constraint-layout/index.html>
- **Medium | Einführung in Android's ConstraintLayout**
<https://medium.com/exploring-android/exploring-the-new-android-constraintlayout-eed37fe8d8f1>
- **Android Developers | LinearLayout**
<https://developer.android.com/reference/android/widget/LinearLayout.LayoutParams.html>
- **Android Developers | Screen Compatibility**
https://developer.android.com/guide/practices/screens_support.html#qualifiers
- **Wikipedia | ISO Language Codes**
https://en.wikipedia.org/wiki/List_of_ISO_639-1_codes
- **Wikipedia | ISO Country Codes**
https://en.wikipedia.org/wiki/ISO_3166-1
- **Android Developers | Best-Matching Resource**
<https://developer.android.com/guide/topics/resources/providing-resources.html#BestMatch>
- **Android Developers | android.widget (View-Klassen APIs Summary)**
<https://developer.android.com/reference/android/widget/package-summary.html>
- **Android Developers | Data Binding Library**
<https://developer.android.com/topic/libraries/data-binding>
- **Android Developers | eigenes CustomToastView Layout**
<https://developer.android.com/guide/topics/ui/notifiers/toasts.html#CustomToastView>
- **Android Developers | DialogFragment**
<https://developer.android.com/guide/topics/ui/dialogs.html#DialogFragment>

9.3 Android 3 - Persistenz

- **Android Developers | Permissions (Übersicht)**
<https://developer.android.com/guide/topics/permissions/overview>
- **Github | PermissionsDispatcher**
<https://github.com/permissions-dispatcher/PermissionsDispatcher>
- **Android Developers | SQLite (*nicht empfohlen*)**
<https://developer.android.com/training/data-storage/sqlite.html>
- **Android Developers | Room**
<https://developer.android.com/training/data-storage/room/index.html>

- **Android Developers | RecyclerView**
<https://developer.android.com/guide/topics/ui/layout/recyclerview>
- **CodeLabs | Android Room with a View(Model)**
<https://codelabs.developers.google.com/codelabs/android-room-with-a-view/#0>
- **Android Developers | Room - Queries in Klassen kapseln**
<https://developer.android.com/training/data-storage/room/creating-views>
- **Android Developers | Room - Observable Queries mit LiveData**
<https://developer.android.com/training/data-storage/room/accessing-data#query-observable>
- **Android Developers | Room - Datenbank migrieren (bspw. bei App-Updates)**
<https://developer.android.com/training/data-storage/room/migrating-db-versions>
- **Android Developers | Room - Datenbank testen**
<https://developer.android.com/training/data-storage/room/testing-db>
- **Android Developers | Room - TypeConverter: Objekt-Referenzen in DB**
<https://developer.android.com/training/data-storage/room/referencing-data>
- **Android Developers | Calendar Provider**
<https://developer.android.com/guide/topics/providers/calendar-provider.html>
- **Android Developers | Contacts Provider**
<https://developer.android.com/guide/topics/providers/contacts-provider.html>

9.4 Android 4 - Kommunikation & Nebenläufigkeit

- **Android Developers | ANR - Keep your App responsive**
<https://developer.android.com/training/articles/perf-anr.html>
- **Android Developers | Background Processing**
<https://developer.android.com/guide/background/>
- **Android Developers | Threading**
<https://developer.android.com/topic/performance/threads>
- **Android Developers | Prozesse & Threads (Übersicht)**
<https://developer.android.com/guide/components/processes-and-threads.html>
- **Android Developers | AsyncTask**
<https://developer.android.com/reference/android/os/AsyncTask?hl=en>
- **Android Developers | Verhaltensänderungen aller Apps**
<https://developer.android.com/about/versions/pie/android-9.0-changes-all?hl=en>
- **HTTP Definition**
<https://tools.ietf.org/html/rfc2616>
- **Github | OkHttpClient**
<http://square.github.io/okhttp/>
- **Github | Retrofit**
<https://square.github.io/retrofit/>
- **JSON Informations**
<http://json.org>
- **JSON Formatter & Validator**
<https://jsonformatter.curiousconcept.com/>
- **Acronime REST Service Doku**
<http://www.nactem.ac.uk/software/acromine/rest.html>

9.5 Android 5 - Services, Broadcast Receiver

- **Android Developers | Services Overview**
<https://developer.android.com/guide/components/services.html>
- **Android Developers | Service**
<https://developer.android.com/reference/android/app/Service.html>
- **Android Developers | BroadcastReceiver**
<https://developer.android.com/reference/android/content/BroadcastReceiver>
- **Android Developers | WorkManager**
<https://developer.android.com/topic/libraries/architecture/workmanager>

9.6 Android 6 - Intents, App-Widgets, Fragments etc.

- **Android Developers | Intents & Intent Filters**
<https://developer.android.com/guide/components/intents-filters.html>
- **Android Developers | PackageManager**
<https://developer.android.com/reference/android/content/pm/PackageManager.html>
- **Android Developers | Fragments**
<https://developer.android.com/guide/components/fragments.html>
- **Android Developers | App Widgets Overview**
<https://developer.android.com/guide/topics/appwidgets/overview>
- **Android Developers | Build an App Widget**
<https://developer.android.com/guide/topics/appwidgets/index.html>
- **Android Developers | Design for Android**
<https://developer.android.com/design/>
- **Material Design**
<https://material.io/design/>
- **Android Asset Studio**
<https://romannurik.github.io/AndroidAssetStudio/index.html>
- **Android Developers | Add the App Bar**
<https://developer.android.com/training/appbar/index.html>
- **Android Developers | Navigation (Wireframing)**
<https://developer.android.com/guide/navigation#wireframe>
- **Android Developers | Support Library**
<https://developer.android.com/topic/libraries/support-library/index.html>
- **Android Developers | Support Library Setup**
<https://developer.android.com/topic/libraries/support-library/setup>
- **Android Developers | Android Jetpack (AndroidX)**
<https://developer.android.com/jetpack>
- **Android Developers | Migrating to AndroidX**
<https://developer.android.com/jetpack/androidx/migrate>
- **Android Developers | Publish your App**
<https://developer.android.com/studio/publish>
- **Android Developers | Launch your app worldwide**
<https://developer.android.com/distribute/best-practices/launch>
- **Android Developers | Alternative distribution options**
<https://developer.android.com/distribute/marketing-tools/alternative-distribution>
- **Android Developers | Launch checklist**
<https://developer.android.com/distribute/best-practices/launch/launch-checklist.html>
- **Wikipedia | Android Application Package**
https://en.wikipedia.org/wiki/Android_application_package
- **ProGuard**
<https://stuff.mit.edu/afs/sipb/project/android/sdk/android-sdk-linux/tools/proguard/docs/index.html#manual/introduction.html>
- **Guardsquare | ProGuard and R8: Comparison of Optimizers**
<https://www.guardsquare.com/en/blog/proguard-and-r8>

9.7 Web- & Hybrid Mobile Apps mit Ionic

- **Ionic | What Is Ionic Framework?**
<https://ionicframework.com/docs/intro>
- **Ionic | Getting Started**
<https://ionicframework.com/getting-started/>
- **Ionic | Installing Ionic**
<https://ionicframework.com/docs/installation/cli>
- **Ionic | Ionic CLI commands**
<https://ionicframework.com/docs/cli>
- **Ionic | ion-header**
<https://ionicframework.com/docs/api/header>
- **Ionic | Network plugin**
<https://ionicframework.com/docs/native/network/>
- **JetBrains | Developing Ionic apps in WebStorm**
<https://blog.jetbrains.com/webstorm/2017/08/developing-ionic-apps-in-webstorm/>
- **NodeJS**
<https://nodejs.org/en/>
- **Apache Cordova | Overview**
<https://cordova.apache.org/docs/en/latest/guide/overview/index.html>
- **Ionic | Ionic Native (Cordova Plugins)**
<https://ionicframework.com/docs/native/>

9.8 Entwicklungsansätze für mobile Applikationen

- **Flutter**
<https://flutter.dev/>
- **Wikipedia | Mobile operating system / Market Share**
https://en.wikipedia.org/wiki/Mobile_operating_system#Market_share
- **Wikipedia | Xamarin**
<https://en.wikipedia.org/wiki/Xamarin>
- **Wikipedia | Just-in-time compilation**
https://en.wikipedia.org/wiki/Just-in-time_compilation
- **Wikipedia | Google Play**
https://en.wikipedia.org/wiki/Google_Play
- **Wikipedia | App Store (iOS)**
https://en.wikipedia.org/wiki/App_Store_%28iOS%29
- **Text**
link